# 基于聚类算法的交通信号灯周期问题的研究

## 摘要

本文针对估计交通信号灯周期问题，通过建立微分方程模型，实现对信号灯周期的预测。

**针对问题一**，本文首先建立了微分方程模型，利用车辆的运动轨迹数据估计出各路口的红灯时长。然后采用K-means聚类算法，估计出各路口的信号灯周期，从而得到各路口的红灯和绿灯时长。

**针对问题二**，本文讨论了样本车辆比例、车流量和定位误差等因素对交通信号灯周期估计的影响。结果表明，这些因素对估计精度产生显著影响，因此在实际应用中需要考虑这些因素，以便更准确地估计信号灯周期。

**针对问题三**，本文建立聚类模型，通过分析聚类结果的变化，快速检测出信号灯周期的变化，并估计出变化后的周期。

**针对问题四**，本文利用蒙特卡洛模拟不同信号灯周期，通过比较不同周期下车辆到达时间的分布情况，估计出最佳的信号灯周期。

综上，本文使用的模型具有以下优点：微分方程模型能准确描述车辆的运动情况；K-means聚类算法能有效判断周期变化；蒙特卡洛模拟能考虑不同周期对车辆到达的影响。这些模型的有机结合，能有效地估计信号灯周期，为导航服务提供支持。

**关键词：交通信号灯周期；微分方程模型；k-means算法；灵敏度分析；蒙特卡洛模型**

# 目录

# 一、问题重述

交通信号灯是交通管理中不可或缺的工具，它的合理运用不仅能有效缓解交通拥堵，提升道路利用率，还能增强交通管理的灵活性，使交通流量控制更加高效。信号灯通过红、黄、绿三种颜色的灯光变化，指示车辆和行人何时停止和何时通行。本题中仅考虑红绿灯周期，即红灯亮时表示停止，此时不允许车辆和行人通过；绿灯亮时表示可以通过。

在本问题中，一家电子地图服务提供商希望能够通过分析其客户车辆的行驶轨迹数据，推测出城市交通信号灯的工作周期。这样的信息将有助于他们为用户提供更精确、更有效的导航服务。

为此，他们收集了以下数据：

1. 附件 1 和附件 2：包含了五个不同路口，在某一方向上，车辆在一小时内的行车轨迹样本。

2. 附件 3：收集了六个路口在单一方向上一小时内车辆行驶轨迹的样本数据。

3. 附件 4：包含了某一特定路口，所有样本车辆的轨迹数据，包括时间、车牌号码和车辆位置等信息。

通过分析这些数据，电子地图服务商希望能够推断出交通信号灯的工作周期，从而优化其导航服务的准确性和实用性。为了能够更准确的计算出信号灯周期，本文需研究以下几个问题：

**问题一：**假设信号灯的周期不会改变，能否通过已知车辆的行车轨迹估算出信号灯的红绿周期。能否建立模型求出各方向的信号灯周期。

**问题二：**只能获取部分使用该产品的行车轨迹，是否会对结果产生显著影响。受各种因素影响，轨迹数据的定位可能会产生误差，此误差是否会对所建立的模型估计的精度产生影响。除此之外，样本车流量，车辆比例是否对估计精度产生影响。

**问题三：**假设信号灯周期会发生改变，能否快速检测出信号灯周期是否发生变化。如果发生了改变，能否计算出变化后的信号灯周期、什么时候发生的变化和变化之前和之后的周期参数。

**问题四：**能否根据某路口所有方向样本车辆的轨迹数据，求出该路口所有信号灯的周期。

# 二、问题分析

## 2.1 对于问题一的分析

由于不考虑该问题信号灯周期的变化，且已知车辆的行车轨迹（但已知车迹的车序并不连续）。我们首先对附件 1 的数据进行缺失值和异常值的检测，发现该附件 1 的所有数据并没有缺失值和异常值。我们利用 x，y 坐标，时间和车辆运动情况建立了微分方程模型，根据该模型推出车辆的运动情况进而可知信号灯的红灯和绿灯状态，再利用不同车在不同时间段计算每个路口红灯的时长，我们用所有车辆路口红灯的时长差值的最大值表示红灯的时长。

然而，在计算过程中我们发现，有些车辆在红灯亮起时并未立即停止。这导致我们的计算结果可能存在较大误差。因此我们将计算出的不同车辆在该路段因红绿灯原因停止运动的最大时间作为最终确定的红灯时长。

对于绿灯时长的计算，我们采用了 K-means 聚类算法来估计一个完整的信号灯周期长度（包括红灯和绿灯的持续时间），再将这个总周期长度中减去红灯的持续时间，得到绿灯的持续时间[1]。通过这种方式，我们可以更准确地估计出每个路口信号灯的红灯和绿灯时长，从而为电子地图服务商提供更精确的数据，以优化其导航服务。

## 2.2 对于问题二的分析

该题依旧不考虑周期变化，且已知车辆行车轨迹（但已知车迹的车序并不连续）。我们先对附件2的数据进行缺失值和异常值的检测，发现数据没有缺失，但出现部分值异常，我们讨论如果剔除异常值后会出现时间轴上的数据缺失，所以我们对该异常值进行处理。用问题一的计算方法去计算处理后的数据，得出不被车辆比例、车流量和定位误差等因素影响的的红绿灯市场，再根据灵敏度分析分别对车辆比例，车流量和定位误差对估计精度的影响。

## 2.3 对于问题三的分析

该问题考虑周期变化的影响，没有其他因素影响数据，且已知车辆行车轨迹。我们首先将附件3中的数据进行缺失值和异常值检测，没有发现数据缺失，但出现数据异常，随后用同种方法处理异常值。再处理完异常值后将原本聚类所分的

簇进行扩大，在扩大到一定的簇类时，再将聚类后的值进行作差处理。若出现差别较大的差值则视为该路口的周期发生了变化，记录周期变化的值，和周期变化时的时刻。

### 2.4 对于问题四的分析

为了准确地分析车辆到达模式，我们使用蒙特卡洛算法模拟信号灯周期。由于无法直接从数据中获得车辆到达模式，我们利用数据集中的时间戳来模拟可能的周期。我们根据问题一至问题三的结果假设一个合理的周期范围为 84 秒到 115 秒。然后，计算每个周期与实际车辆到达时间的偏差，以评估其匹配程度。通过比较每个周期内车辆到达的平均时间间隔，我们检查这些间隔是否显示出规律性，从而估计最佳信号灯周期。

## 三、模型假设

假设一：假设该车辆行驶过车中均遵守交通规则。
假设二：假设该检测路段并没有发生影响重大的交通事故。
假设三：假设信号灯未损坏，均正常。
假设四：假设该路口车道较宽，各车辆行驶间互不影响（不堵车）。
假设五；假设如果有异常值，则该异常值的数据均在可计算分析的误差内。

## 四、符号说明

符号说明要点如下：

| 符号 | 说明 |
| --- | --- |
| $x_i$ | 某一辆车在第 i 秒的 x 轴位置 |
| $y_i$ | 某一辆车在第 i 秒的 y 轴位置 |
| $t_i$ | 第 i 秒的时刻 |
| $v_i$ | 第 i 秒的瞬时速度 |

# 五、模型建立

## 5.1 问题一模型的建立

### 5.1.1 问题一模型的准备

对附件 1 中的样本数据进行异常值和缺失值检测，由图 1 可知并没有缺失值和异常值[2]。

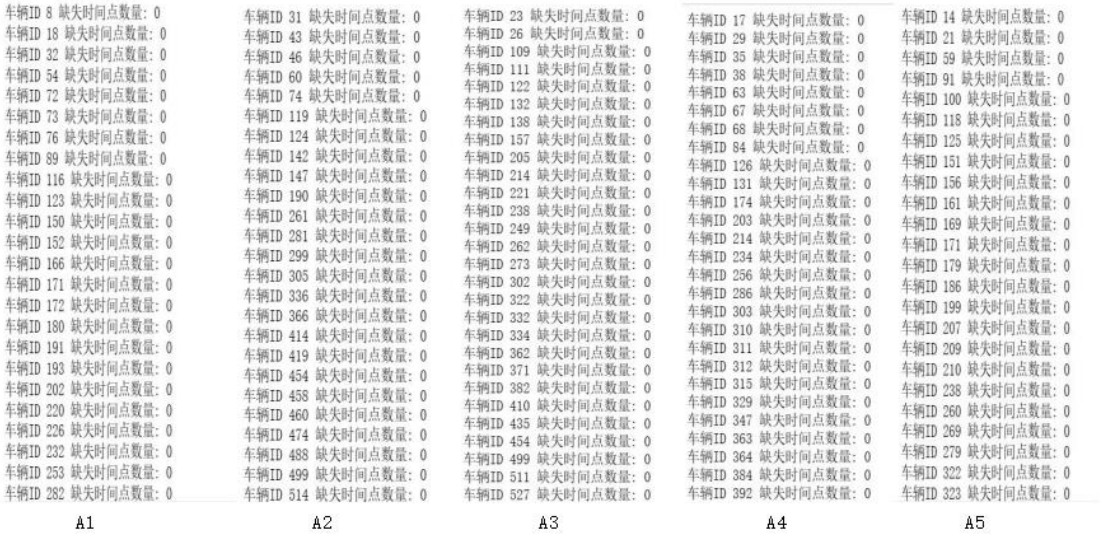


图 1 附件 1 中数据的缺失值个数

### 5.1.2 模型的建立

**微分方程模型**

根据部分车的运动情况可以大致推断信号灯情况，故此建立此方程模型。

$$v_i = \frac{\sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}}{t_{i+1} - t_i}$$

根据此模型算出的速度，令大于五个时间点的速度接近零时的原因是该车开始等红灯，然后根据检测车的等待时长的最大值当作红灯时长。

**K-means 算法**

K均值聚类算法（k-means clustering algorithm）是一种基于迭代求解的聚类分析算法，通过迭代过程将数据集划分K个类别，使得每个类别内的数据点之间的相似度最高，而类别间的相似度最低，具体步骤如图2所示。

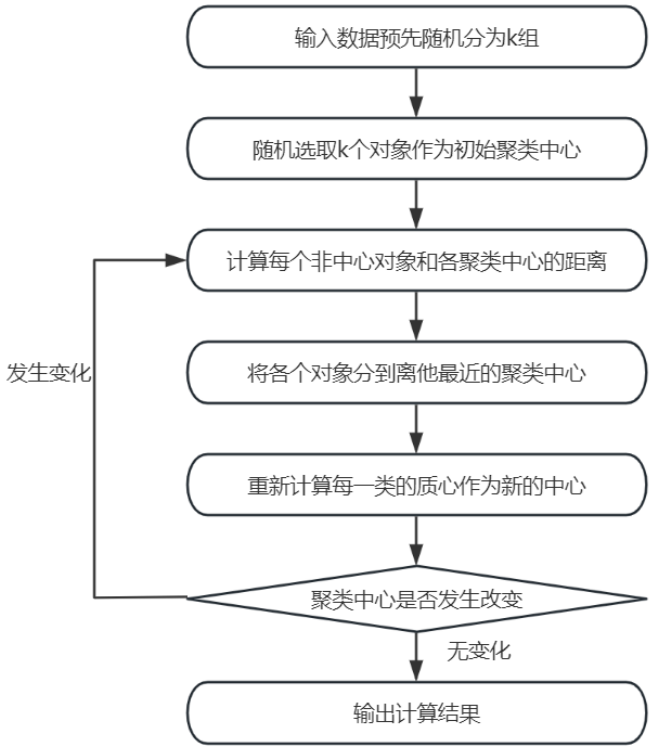


**图1　k 均值聚类算法流程图**

在对附件A1至A5的数据进行分析时，我们首先做出了一个假设，即所有车辆都遵守交通规则，没有闯红灯等违法行为。基于这个假设，我们使用建立的微分方程模型来计算红灯的持续时间。

### 5.1.3模型的求解

在本题中，利用每辆车等完红灯的时刻为一个总周期的起始点到下一辆检测车等完红灯时刻为下一个总周期的起始点，用下一个总周期的起始点减去上一个周期的起始点得出至少一个的完整周期。然后进行K-means聚类，再将每个簇密集的中心点做差得出一个总周期的时长。

我们首先计算出一个车辆的大致运动情况，用x，y算出该车每秒的欧式距离位移，再用已知位移计算该车辆每秒的速度，根据红灯停绿灯行的交通法则可知，如果该车速度为零或者小于一个阈值（这个阈值由正常青年跑1km时的速度定义），则说明该车就是在等红灯。根据该模型推出车辆的运动情况进而可知信号

灯的红灯和绿灯状态，再利用不同车在不同时间段计算每个路口红灯的时长，我们用所有车辆路口红灯的时长差值的最大值表示红灯的时长。

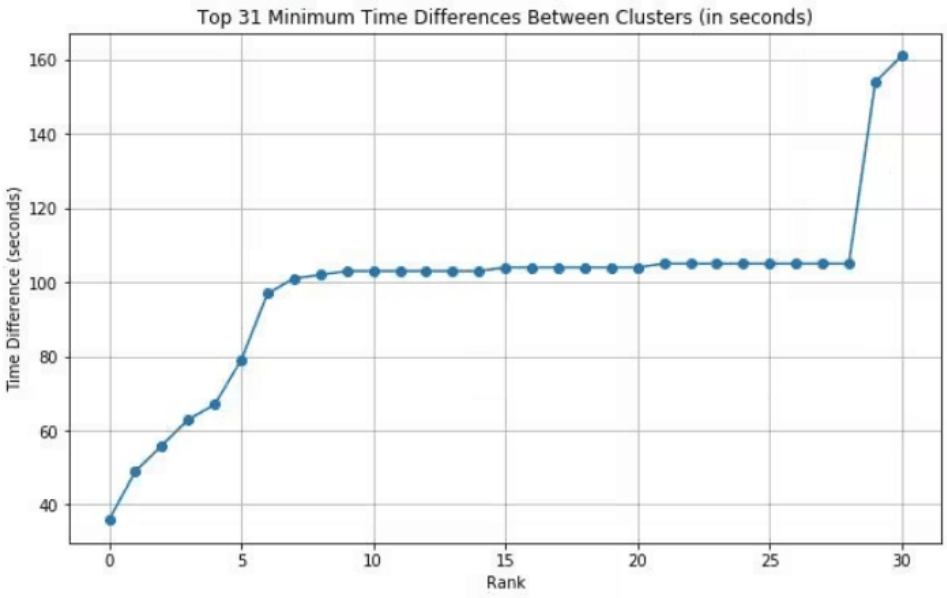


图 2　问题一聚类后的簇之间时间差的散点图

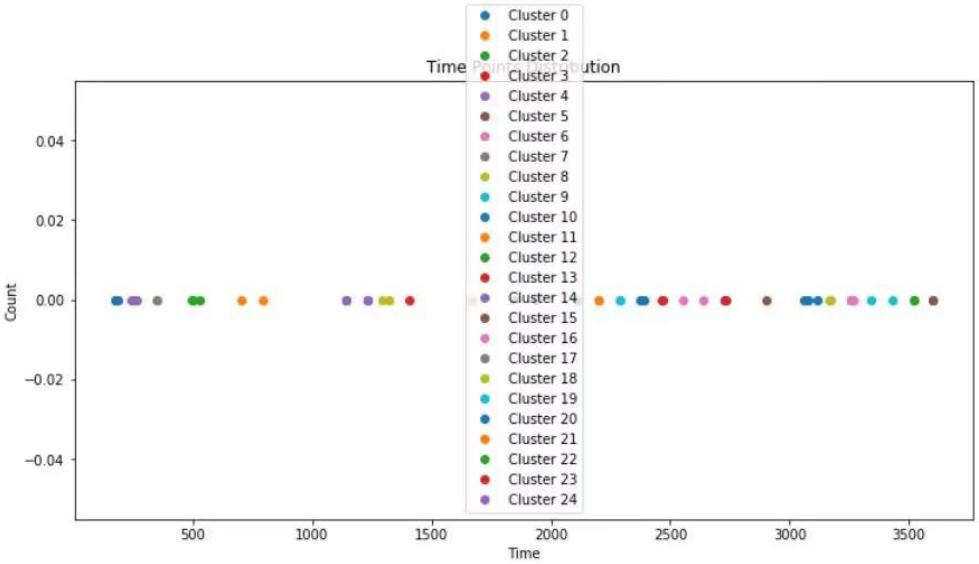


图 3　各聚类中心大小散点图

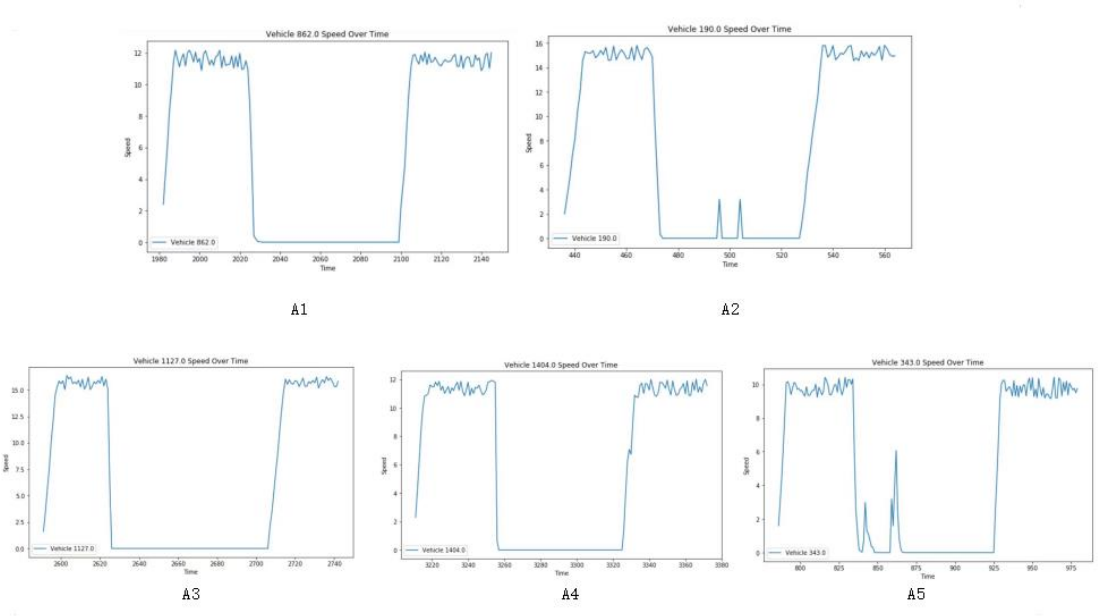


图 4　问题一中各路口样本车辆最大的速率图

然而，在计算过程中我们发现，有些车辆在红灯亮起时并未立即停止（该车还未到达信号灯）如图 5。这导致我们的计算结果可能存在较大误差。因此我们将计算出的不同车辆在该路段因红绿灯原因停止运动的最大时间作为最终确定的红灯时长，并在图像上进行人工筛选出正常符合实际情况的 A2,A5 路口的速度与时间图像，如图 6。

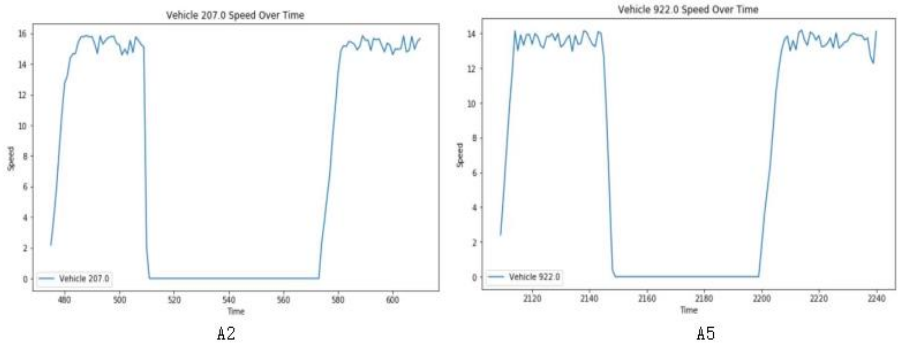


图 5　停止运动的最大时间的 speed-time 图

在计算信号灯的总周期时长时，我们首先采用了 K-means 聚类算法。但在应用 K-means 算法之前，我们使用了轮廓系数来评估数据的分类情况。轮廓系数是一种用于衡量聚类质量的方法，它可以帮助我们确定最佳的聚类数，从而更准确地应用 K-means 算法来估计信号灯的完整周期时长，包括红灯和绿灯的持续时间[3]。

表 1 附件 1 中 A1-A5 路口的轮廓系数

| 路口 | 轮廓系数 |
|---|---|
| A1 | 0.892 |
| A2 | 0.748 |
| A3 | 0.869 |
| A4 | 0.798 |
| A5 | 0.794 |

我们发现表 1 中轮廓系数均大于 0.7（轮廓系数越接近 1，表示簇内的密集度越高），说明选取的算法比较合适。

由于不是所有车辆都会在红灯开始时到达路口，我们采取了以下策略：我们查找了所有车辆在路口等待红灯的时间，并记录下这些等待时间的最大值，将这个最大值作为红灯时长的估计。

在计算绿灯时长时，我们不能简单根据车辆行驶的时间来估计。因此，我们转向分析车辆结束等待红灯的时刻。我们使用 K-means 聚类算法来对这些时间差进行分类，这些时间差代表了车辆从一个红灯周期结束到下一个红灯周期开始的时间段。由于这些时间差可能并不完全相同，我们利用轮廓系数来评估聚类的效果，轮廓系数越接近 1，说明聚类效果越好。通过这种方法，我们确定了最适合该路口时间差分类的 k 个簇。

一旦我们确定了 k 个簇，我们就可以通过计算相邻簇的聚类中心值之间的差异来估计一个完整的信号灯周期（包括红灯和绿灯）。然后，我们从总周期中减去红灯时长，得到绿灯时长的估计。

最终，我们得出了下表 2 所示的结果，这些结果反映了我们对路口信号灯周

期的分析和估计。

表 2　A1-A5 路口的信号灯周期

| | A1 | A2 | A3 | A4 | A5 |
|---|---|---|---|---|---|
| 红灯 | 68 | 50 | 80 | 68 | 62 |
| 绿灯 | 36 | 38 | 23 | 18 | 26 |

## 5.2　问题二的模型建立

### 5.2.1 问题二的模型准备

建立模型前，首先进行了数据的预处理，对缺失值和异常值进行检测。结果如图 6 和图 7 所示。由此发现，样本中存在部分异常值。在对异常值进行了插值处理后，我们运用了问题一所建立的模型进行分析。

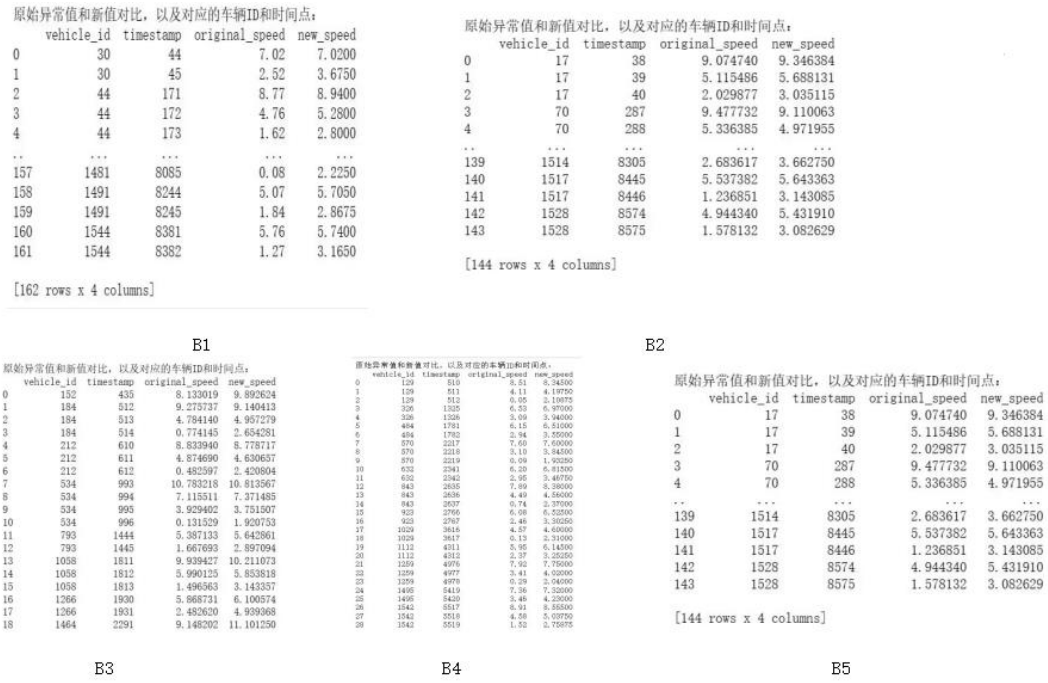


图 6 附件二中数据的缺失值个数

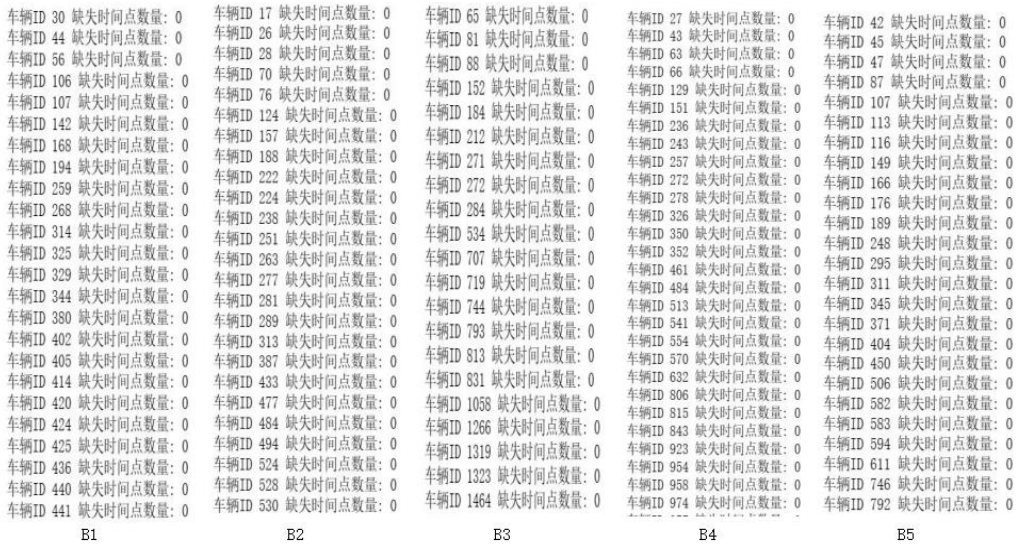


**图 7　异常值的查找和插值处理**

### 5.2.2 模型的建立

在解决这个问题时，我们采用了基于问题一所建立的模型的修改版本。我们仍然假设，当车辆的速度在连续五个时间点内接近零时，这表明车辆开始等待红灯。我们记录下这些等待时间，并将最长等待时间作为红灯的持续时间。

为了确定信号灯的完整周期，我们采用了一种方法：选择一辆车在等待红灯之后开始行驶的时刻作为周期的起始点，并记录下一辆车在等待红灯结束后开始行驶的时刻作为下一个周期的起始点。通过计算这两个时刻之间的时间差，我们可以估计出信号灯的周期长度。

在应用这个模型时，我们还使用了 K-means 聚类算法来确定最佳的周期划分，并通过计算轮廓系数来评估聚类的质量。轮廓系数的正常值范围表明我们的聚类效果是合理的，模型表现良好。

用 K-means 算法算出一个总周期的时长。根据处理完异常值后的正常数据计算出无误差，无影响的正常红绿灯时长，再根据灵敏度分析。分别改变车辆比例，车流量和定位误差的值，来判断该变量对估计精度的影响。

### 5.2.3 模型的求解

在分析路口红绿灯变化趋势时，我们改变了车辆比例，分别计算了当车辆比例为 20%，40%，60%，80%时路口红绿灯的时长变化。我们观察图 9 可知，随

着车辆比例的减少，红灯时长也呈现出减少的趋势。我们推测这种变化的原因可能是，我们计算红灯时长的模型是基于路口所有等待红灯的车辆的等待时间的最大值。当车辆比例减少时，这个最大值的误差可能会增大，导致我们估计的最大值变小，从而使红灯时长的估计值减小。由于车辆比例的变化对总体的聚类分析影响较小，总周期不会因为比例的变化而有显著变化。因此，由于绿灯时长是从总周期时长中减去红灯时长得到的，所以绿灯时长的估计值会增大。

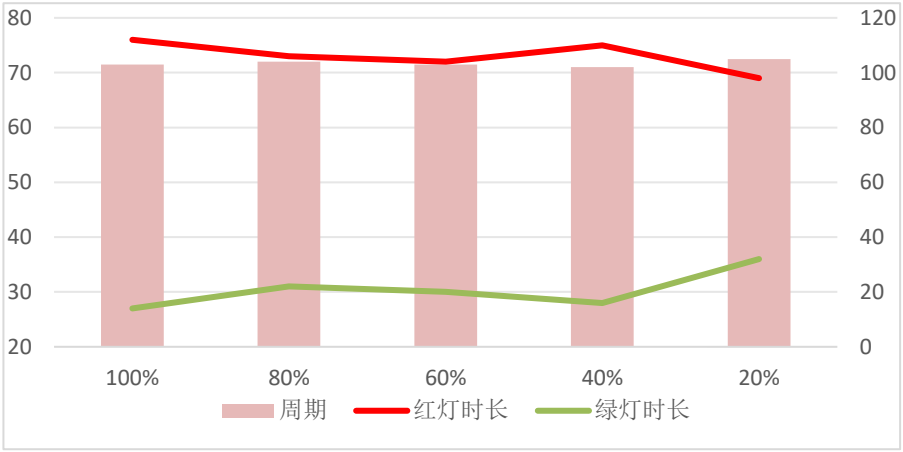


图 8　样本车辆比例对红绿灯周期的影响

如图 10 的数据处理分析来说，随着车流量的增加，信号灯周期的估计可能变得更加困难，因为车辆停留模式可能更加复杂。因此，在实际应用中，需要考虑车流量对模型估计精度的影响，并根据这些分析结果来调整模型或数据处理方法，以提高信号灯周期估计的准确性。结果如表明，车流量对信号灯周期估计具有显著影响。

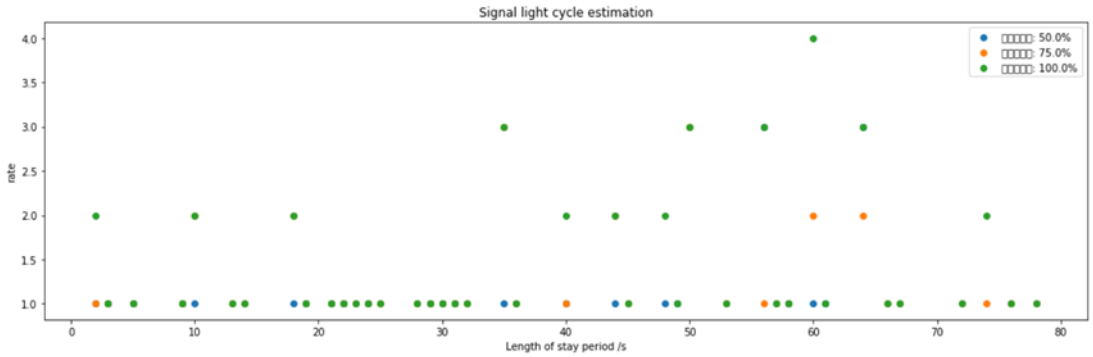


图 9 车流量对估计精度的影响

该问题的建模方法主要是基于数据的统计分析，通过处理车辆的位置数据来估计停留时间，并分析定位误差对此估计的影响。通过处理车辆位置数据、计算移动距离，并设置阈值来判断车辆是否停止，进而估计车辆的停留时间。随后，模拟不同的三个定位误差水平，通过比较原始数据和含误差数据的停留持续时间频率分布，分析了定位误差对停留时间估计的影响，旨在评估模型的鲁棒性和准确性，具体结果如下：

| | time | vehicle_id | x | y | distance |
|---|---|---|---|---|---|
| **0** | 69 | 30 | -494.90 | -4.8 | <NA> |
| **1** | 70 | 30 | -492.79 | -4.8 | 2.11 |
| **2** | 71 | 30 | -489.35 | -4.8 | 3.44 |
| **3** | 72 | 30 | -483.59 | -4.8 | 5.76 |
| **4** | 73 | 30 | -475.96 | -4.8 | 7.63 |

图 10 连续的停止序列

| | <lambda> |
|---|---|
| vehicle_id | |
| **30** | [115, 116, 117, 118, 119, 120, 121, 122, 123, ... |
| **44** | [139, 140, 141, 142, 143, 144, 145, 146, 147, ... |
| **142** | [367, 368, 369, 370, 371] |
| **168** | [436, 437, 438, 439, 440, 441, 442, 443, 444, ... |
| **259** | [645, 646, 647, 648, 649, 650, 651, 652, 653, ... |

图 11 停止持续时间的频率分布

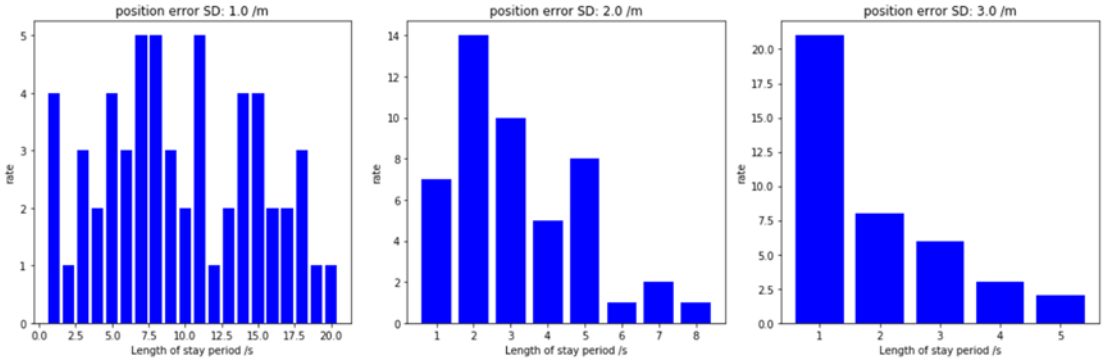


图 12 各个标准差的频率分布图

从图 11-13 所示，我们可以推断定位误差对模型精度具有显著影响。

具体来说，随着定位误差的增加，车辆停留持续时间的分布发生了变化。这表明在实际停留时间与模型预测的停留时间之间存在差异，可能导致对交通状况的误判，例如错误地将车辆分类为长时间或短时间停留。

此外，不同定位误差水平下停留持续时间的频率分布变化，进一步表明定位误差可能对车辆停留行为的统计分析产生显著影响，从而影响基于这些数据的进一步分析和决策过程。因此，在使用基于位置数据的模型时，特别是在需要精确停留时间估计的应用场景中，如交通流量分析、城市规划或车辆管理系统，必须考虑定位误差的影响，定位误差可能会对模型的准确性和可靠性产生重大影响。

我们在对 B3 路口进行聚类分簇时发现该路口的簇值小，聚类出的数目也少，故此我们分析造成该现象的原因是聚类值相差较大。

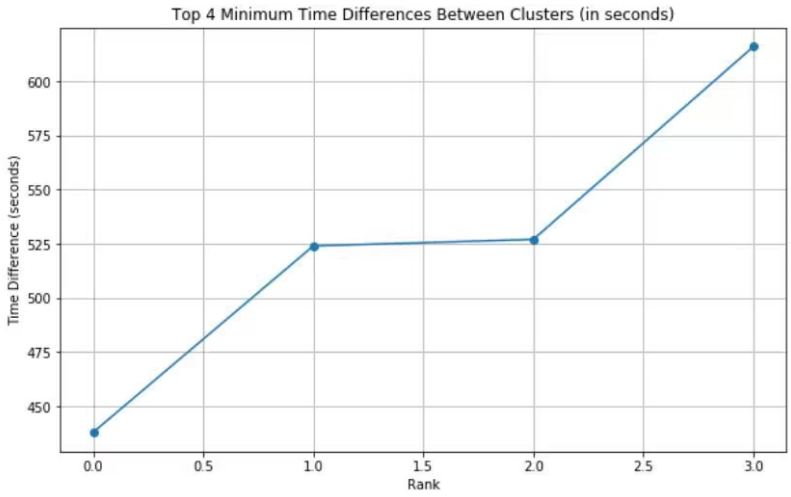


**图 13　集群间的最小时间差**

当聚类数目本身较小，聚类的数目也少，计算时间点差值得到周期的 k 倍（k≠1）时。

综上我们得出最终结果如下表：

**表 3　B1-B5 的红绿灯周期时长**

| 路口 | B1 | B2 | B3 | B4 | B5 |
|---|---|---|---|---|---|
| 红灯（秒） | 76 | 79 | 70 | 77 | 95 |
| 绿灯（秒） | 27 | 34 | 18 | 28 | 20 |

## 5.3 问题三的模型建立

### 5.3.1 问题三的模型准备

基于对问题二异常值的处理方式，同样用在问题三中，无缺失值，出现部分异常值，在修改完异常值后再进行后续的模型建立或者思路算法。

### 5.3.2 模型的建立

在问题一的模型基础思路上，我们对该模型进行部分的修改。为了保证既能合理的聚类，又要区分出周期的变化。故此我们对原本数据先转变为问题一中的速度数据，根据速度大小开始对时间进行 K-means 聚类，与之前不同的是，此处的 K-means 聚类作差进行俩次，才能判别出周期是否发生变化，和发生变化的时刻。

### 5.3.3 模型的求解

在进行第一次的 K-means 聚类得出的六个路口的一个总周期时长的结果为图 15 和 16 所示。

我们发现除 C3 外其他路口的这个周期时长并不是一个周期的，故此我们把周期时长不相等时认为没有发生了周期的变化，且该些点的周期时长是 n 倍的周期时长，所以只有 C3 中出现两个周期不是倍数关系的周期时长，且数据合理正常。

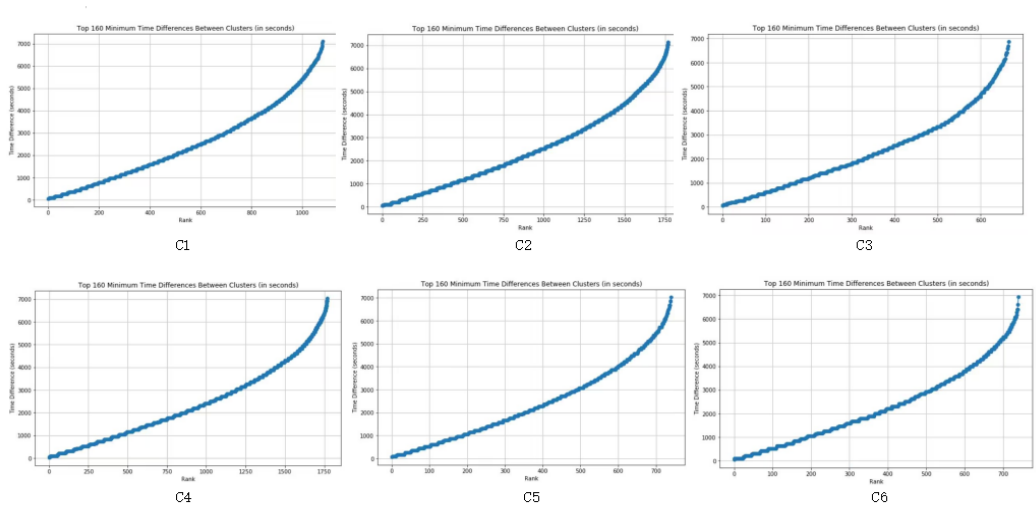


图 14 各路口某样本车辆的速度随时间变化图

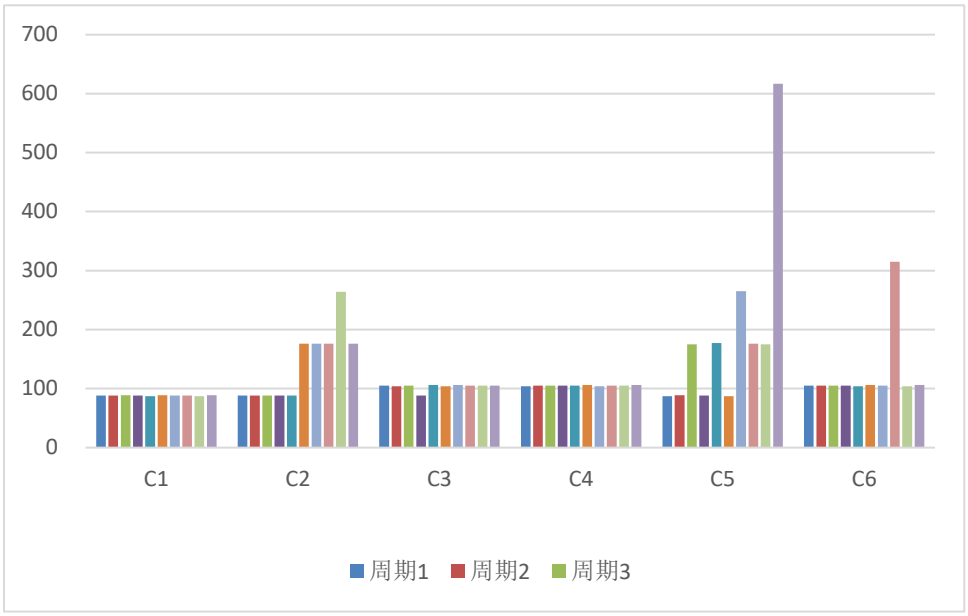


图 15　问题三中各路口的信号灯周期

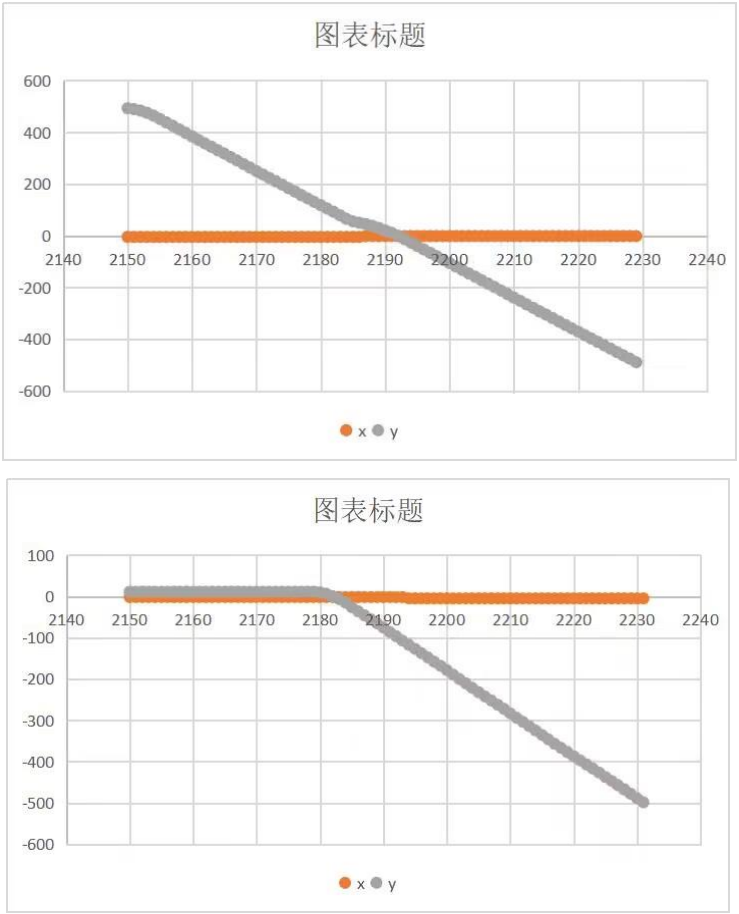


图 16　附件三 C3 文件数据处理中发生改变的周期

在得知两个周期值后，我们发现那个周期时长较小的值只出现一次，我们就用该时长带入数据中得出一个该周期时长时间范围，在该范围中发现在周期时长内只有三辆车的数据，如图 16，17 所示，且有一辆车在到信号灯之前信号灯结束红灯。我们把另外两个车的等红灯的时间放大发现有一辆车在更早之前开始等红灯。我们就把该车开始等红灯的时间当成接近红灯周期变化后的最初时间。

最后根据计算得出表 4：

**表 4　六个路口的周期切换时刻表**

| 路口 | C1 | C2 | C3 | C4 | C5 | C6 |
|---|---|---|---|---|---|---|
| 周期 1 红灯的时长（秒） | 55 | 66 | 77 | 73 | 55 | 76 |
| 周期 1 绿灯的时长（秒） | 33 | 22 | 33 | 32 | 33 | 28 |
| 周期切换时刻 | 无 | 无 | 2129 | 无 | 无 | 无 |
| 周期 2 红灯的时长（秒） | | | 50 | | | |
| 周期 2 红灯的时长（秒） | | | 38 | | | |
| 周期切换时刻 | | | 2217 | | | |
| 周期 3 红灯的时长（秒） | | | 77 | | | |
| 周期 3 红灯的时长（秒） | | | 33 | | | |

## 5.4　问题四的模型建立

### 5.4.1 模型的准备

根据对数据集中的数据进行分析，我们认为不存在缺失值，出现部分异常值，在修改完异常值后再进行后续的模型建立或者思路算法。

### 5.4.2 模型的建立

该问题类似与蒙特卡洛模型，蒙特卡洛方法是一种基于概率和统计理论的数值计算方法，它的核心思想是通过使用随机数（或随机变量）来解决计算问题，特别是那些通过确定性的方法来难以解决的问题。蒙特卡洛方法通过模拟大量的随机试验来获得问题的数值解或概率解。

### 5.4.3 模型的求解

据分析，我们最终并未直接建立传统的数学模型，如线性回归、分类器等，而是通过一系列数据处理和分析步骤来估计信号灯的最佳周期。

首先，我们在 kaggle 中读取附件 4 中“D.csv”数据集中的轨迹数据，将其

转换为后续可以操作的数据结构，具体如图 18 所示，为下一步决定如何进一步处理它做准备。

| | time | vehicle_id | x | y |
|---|---|---|---|---|
| 0 | 0 | 0 | 48.27 | 496.27 |
| 1 | 1 | 0 | 48.09 | 494.76 |
| 2 | 2 | 0 | 47.65 | 491.09 |
| 3 | 3 | 0 | 46.92 | 484.99 |
| 4 | 3 | 2 | 495.39 | -21.70 |

图 17　附件 4 中“D.csv”数据集中的轨迹数据

为了分析车辆到达模式，我们需要从位置数据中确定每辆车的到达和离开路口时间及其行驶方向。由于数据中未直接提供行驶方向，我们考虑基于位置变化来估计。首先，我们尝试通过计算速度来识别接近路口的车辆，但数据中未发现速度较慢的点。因此，我们改用距离阈值法，通过计算车辆到假设路口位置的距离来筛选接近路口的车辆，但此方法也未有效筛选出数据点。最后，我们通过观察车辆位置是否在短时间内发生显著变化来识别接近或离开路口的行为。这种方法帮助我们找到了一些位置变化显著的车辆，推测这些车辆可能在接近或离开路口。

```
Empty DataFrame
Columns: [time, vehicle_id, x, y, speed]
Index: []
Empty DataFrame
Columns: [time, vehicle_id, x, y, speed, distance_to_intersection]
Index: []
```

图 18　用速度和距离阈值法计算出的结果

```
   time  vehicle_id       x       y       speed  distance_to_intersection  \
3     3           0   46.92  484.99    6.143525                131.758805
4     3           2  495.39  -21.70  676.653602                544.973470
5     4           0   45.95  476.93  671.288456                126.710369
6     4           2  492.98  -21.56  669.573074                543.286964
7     5           0   44.78  467.16  663.121767                121.034972

   delta_x  delta_y  abs_delta_x  abs_delta_y
3    -0.73    -6.10         0.73         6.10
4   448.47  -506.69       448.47       506.69
5  -449.44   498.63       449.44       498.63
6   447.03  -498.49       447.03       498.49
7  -448.20   488.72       448.20       488.72
```

图 19　观察车辆位置变化显著值所计算出的结果

为了准确地分析车辆到达模式，我们使用蒙特卡洛算法模拟信号灯周期。由于无法直接从数据中获得车辆到达模式，我们利用数据集中的时间戳来模拟可能的周期。我们根据问题一至问题三的结果假设一个合理的周期范围为 84 秒到 115 秒。然后，计算每个周期与实际车辆到达时间的偏差，以评估其匹配程度。通过比较每个周期内车辆到达的平均时间间隔，我们检查这些间隔是否显示出规律性，从而估计最佳信号灯周期。

但在计算匹配程度得分时，较短周期内的平均时间间隔自然会更接近周期的一半，从而产生较低的得分。另一种可能的方法是考虑车辆到达时间间隔在整个周期内的分布，而不是仅仅计算它们与周期中点的接近程度。但以上两种方法都没有正确地反映周期与实际到达模式之间的关系，会导致匹配程度得分总是倾向于周期范围的最大值或最小值。

为了更准确地估计信号灯周期，我们重新考虑匹配程度得分的计算方法。考虑到熵是一种衡量分布无序程度的指标，熵值越低，表示分布越有序。在这个情况下，我们可以使用熵来评估车辆到达时间间隔在周期内的分布情况。

```
   vehicle_id  min  max
0           0    3   74
1           2    3  158
2           4    6  152
3           6    8   85
4           8   13  117
Best cycle (entropy method, all data): 98.0
```

图 20　用熵值估计车辆时间间隔内的周期分布

最终结果“Best cycle (entropy method, all data): 98.0”表示，在所有模拟的周期中，周期为 98 秒时，整个数据集的到达时间间隔分布最为有序。这意味着在这个周期下，车辆到达路口的时间间隔更加规律，可能更符合信号灯的工作周期。

# 六、模型评价

## 6.1 模型的优点

微分方程模型能够精确地描述系统的动态变化，包括系统的演化、增长、衰减等。微分方程模型可以同时考虑多个变量之间的关系，适合于多因素影响的复杂系统。

轮廓系数的取值范围是-1 到 1，值越接近 1 表示聚类效果越好，值越接近-1 表示聚类效果越差。代码中计算了 k 从 2 到 10 时的轮廓系数，并选择轮廓系数最高的 k 作为最优聚类数。当输出 Optimal number of clusters: 3 时，这意味着在计算的范围内，k=3 时得到的聚类结果在所有计算中具有最高的轮廓系数，因此被认为是最优的聚类数。简单来说，根据轮廓系数的评价标准，将数据聚为 3 类可以得到最佳的聚类效果，如图 22。

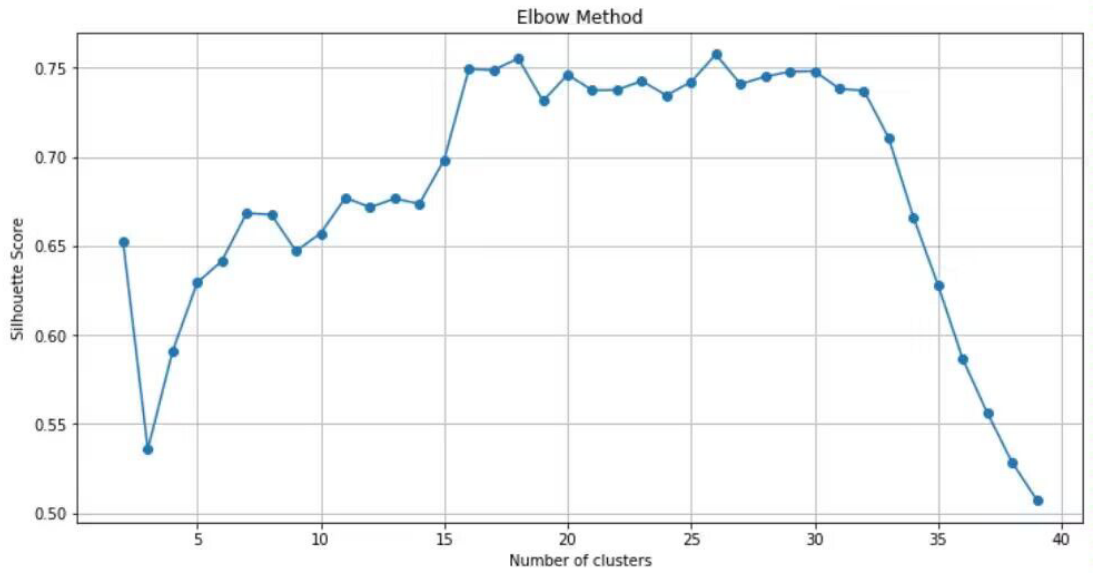


图 21 肘部法则图

在问题四中，我们对模型进行了优化。为了更为准确地估计信号灯周期，也可以引入更复杂的统计方法或考虑其他相关因素，如不同方向的车辆流量、信号灯的相位设置等。

由于我们考虑的情况比较理想，可能需要根据实际数据的特点进行调整。此

外，信号灯周期的识别通常涉及更复杂的交通工程知识，可能需要考虑多种因素，如交通密度、行人过街需求、相邻交叉口的信号配时等。因此，实际应用中可能需要与交通工程师或规划专家合作，以确保分析的准确性和实用性。

### 6.2 模型的缺点

1. 微分方程模型的解依赖于初始条件，初始条件的微小差异可能导致最终结果的显著不同。

2. 微分方程模型往往需要复杂的数学计算，很容易计算错误。

3. 部分该范围时间内攫取的车辆数据较少，后期车辆数据的红灯时长计算的精度受车数量的影响较大，数据较少时，误差较大。

## 八、模型推广

本文主要应用了用于描述系统随时间或空间变化的动态行为的微分方程模型，该模型可以快速计算出样本车辆的运动情况，从而计算出信号灯周期，可以应用于导航服务，规划出避免堵车和等红灯的路线，使人们的出行变得更加方便。同时，此模型还可以应用于生物学，医学等领域，用来描述神经网络的动态变化等等。

## 九、参考文献

[1] 郝美薇, 戴华林, 郝琨. 基于密度的 K-means 算法在轨迹数据聚类中的优化[J]. 计算机应用, 2017, 37 (10): 2946-2951.

[2] 詹鹏宇. 基于车辆轨迹数据的路网优化方法研究[D]. 北方工业大学, 2019.

[3] 赵肄江, 方辰昱, 廖祝华. 一种基于轨迹数据的红绿灯位置检测方法[J]. 测绘地理信息, 2024, 49 (02): 122-130. DOI:10.14188/j.2095-6045.2022657.

# 十、附录

## 附录一　问题一代码（python）

```
# 读取数据集
# 导入相关库
import pandas as pd
import numpy as np
# 读取 CSV 文件
data = pd.read_csv(r"C:\Users\Administrator\Desktop\B 题\附件\附件 1\A1.csv")
# 打印前五行
print(data.head())

#缺失值处理
# 确保时间点是整数
data['time'] = data['time'].astype(int)
# 按车辆 ID 分组，并对每个分组进行处理
grouped = data.groupby('vehicle_id')
# 创建一个空的 DataFrame 来存储处理后的数据
processed_data = pd.DataFrame()
# 对每个分组进行处理
for group_id, group in grouped:
    # 获取当前分组的时间点列表
    time_points = group['time'].tolist()
    # 生成完整的时间点列表
    full_time_points = list(range(time_points[0], time_points[-1] + 1))
    # 找出缺失的时间点
    missing_time_points = set(full_time_points) - set(time_points)
    # 统计缺失的时间点数量
    missing_count = len(missing_time_points)
    # 打印缺失时间点数量
    print(f"车辆 ID {group_id} 缺失时间点数量: {missing_count}")
    # 对于每个缺失的时间点，创建一个新的行
    for missing_time in missing_time_points:
        new_row = {'time': missing_time, 'vehicle_id': group_id, 'x': 0, 'y': 0}
        processed_data = processed_data.append(new_row, ignore_index=True)
    # 将当前分组的数据添加到处理后的数据中
    processed_data = processed_data.append(group, ignore_index=True)
# 对处理后的数据进行排序
processed_data.sort_values(by=['vehicle_id', 'time'], inplace=True)
# 重置索引
```

```
processed_data.reset_index(drop=True, inplace=True)
# 将处理后的数据保存到新的 CSV 文件
processed_data.to_csv(r"C:\Users\Administrator\Desktop\B 题\附件\附件 1\processed_A1.csv",
index=False)
# 异常值处理
import pandas as pd
import numpy as np
from scipy.interpolate import interp1d
# 读取 CSV 文件
data = pd.read_csv(r"C:\Users\Administrator\Desktop\B 题\附件\附件 1\processed_A1.csv")
# 根据车辆 ID 分组数据
grouped_data = data.groupby('vehicle_id')
# 初始化一个空 DataFrame 来存储结果
velocities = pd.DataFrame(columns=['vehicle_id', 'time', 'speed'])
# 为每辆车计算速度
for vehicle_id, group in grouped_data:
    # 计算 x 和 y 坐标、时间的变化
    group['delta_x'] = group['x'].diff()
    group['delta_y'] = group['y'].diff()
    group['delta_t'] = group['time'].diff()
    # 计算相邻点之间的欧氏距离（速度）
    group['speed'] = ((group['delta_x']**2 + group['delta_y']**2)**0.5) / group['delta_t']
    # 移除速度为 NaN 的行
    group = group.dropna(subset=['speed'])
    # 使用.loc[]或.iloc[]来追加结果，而不是.append()
    velocities = velocities.append(group[['vehicle_id', 'time', 'speed']].loc[group.index])
# 计算速度的四分位数范围（IQR）
Q1 = velocities['speed'].quantile(0.25)
Q3 = velocities['speed'].quantile(0.75)
IQR = Q3 - Q1
# 定义异常值边界
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
# 检测速度中的异常值
outliers = velocities[(velocities['speed'] < lower_bound) | (velocities['speed'] > upper_bound)]
# 如果没有检测到异常值，则直接使用原始数据
if outliers.empty:
    # 直接使用原始数据
    interpolated_data = velocities.copy()
    print("不存在异常值")
else:
    # 对异常值进行插值处理
```

```
    interpolated_data = pd.DataFrame(columns=['vehicle_id', 'time', 'x', 'y'])
    for vehicle_id, group in outliers.groupby('vehicle_id'):
        # 计算 x 和 y 坐标的线性插值函数
        x_interp = interp1d(group['time'], group['x'], kind='linear')
        y_interp = interp1d(group['time'], group['y'], kind='linear')
        # 计算异常值点的插值坐标
        interpolated_x = x_interp(group['time'])
        interpolated_y = y_interp(group['time'])
        # 创建插值坐标的新 DataFrame
        interpolated_group = pd.DataFrame({
            'vehicle_id': vehicle_id,
            'time': group['time'],
            'x': interpolated_x,
            'y': interpolated_y
        })
        # 将插值数据追加到 interpolated_data DataFrame
        interpolated_data = interpolated_data.append(interpolated_group)
# 显示插值后的 DataFrame
print(interpolated_data)
# 确保将以下路径替换为您实际想要保存的路径
output_file_path = r"C:\Users\Administrator\Desktop\B 题\附件\附件 1\processed1_A1.csv"
# 将插值后的数据保存到新的 CSV 文件
interpolated_data.to_csv(output_file_path, index=False)
# 红灯时长
# 导入相关库
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
# 读取 CSV 文件
data = pd.read_csv(r"C:\Users\Administrator\Desktop\B 题\附件\附件 1\processed1_A1.csv")
# 打印前五行
print(data.head())
# 按车辆 ID 分组数据
grouped_data = data.groupby('vehicle_id')
# 随机选择五辆车
random_vehicles = grouped_data.apply(lambda x: x.sample(1)).reset_index(drop=True)
# 绘制图像
for index, row in random_vehicles.iterrows():
    vehicle_id = row['vehicle_id']
    group = data[data['vehicle_id'] == vehicle_id]
    plt.figure(figsize=(10, 6))  # 创建一个新的图形窗口
    plt.plot(group['time'], group['speed'], label=f'Vehicle {vehicle_id}')
```

```
        plt.title(f'Vehicle {vehicle_id} Speed Over Time')
        plt.xlabel('Time')
        plt.ylabel('Speed')
        plt.legend()
plt.show()
import pandas as pd
# 读取 CSV 文件
data = pd.read_csv(r"C:\Users\Administrator\Desktop\B 题\附件\附件 1\processed1_A1.csv")
# 初始化一个空 DataFrame 来存储结果
df1 = pd.DataFrame(columns=['vehicle_id', 'interval-red'])
# 按车辆 ID 分组数据
grouped_data = data.groupby('vehicle_id')
# 筛选出同一辆车的记录
for vehicle_id, group in grouped_data:
    # 筛选出速度为 0 的点
    zero_speed_points = group[group['speed'] == 0]
    # 检查是否有不少于五个时间点的速度为 0
    if zero_speed_points.shape[0] >= 5:
        # 计算最后一个速度为零的时间点的时间和第一个速度为零的点的时间差
        last_zero_speed_time = zero_speed_points['time'].iloc[-1]
        first_zero_speed_time = zero_speed_points['time'].iloc[0]
        interval_red = last_zero_speed_time - first_zero_speed_time
        # 将结果添加到 df DataFrame
        df1 = df1.append({'vehicle_id': vehicle_id, 'interval-red': interval_red},
ignore_index=True)
    else:
        # 如果少于 5 个时间点，则 interval-red 记为 0
        df1 = df1.append({'vehicle_id': vehicle_id, 'interval-red': 0}, ignore_index=True)
# 显示结果
print(df1)
# 确保将以下路径替换为您实际想要保存的路径
output_file_path = r"C:\Users\Administrator\Desktop\B 题\附件\附件 1\processed2_A1.csv"
# 将结果保存到新的 CSV 文件
df1.to_csv(output_file_path, index=False)
# 找出 interval-red 最大值的车辆 ID 以及结果
max_interval_red = df1['interval-red'].max()
max_vehicle_id = df1[df1['interval-red'] == max_interval_red]['vehicle_id'].iloc[0]
max_vehicle_id, max_interval_red
print("红灯时长为",max_interval_red)

# 聚类分析
import pandas as pd
```

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
import matplotlib.pyplot as plt
import numpy as np
# 读取 CSV 文件
data = pd.read_csv(r"C:\Users\Administrator\Desktop\B 题\附件\附件 1\processed1_A1.csv")
# 检查 CSV 文件中的列名
print(data.columns)
# 将时间转换为时间戳
data['time'] = pd.to_datetime(data['time'], unit='s')  # 假设时间是以秒为单位
# 找出速度从非零变为零的点
stops = data[(data['speed'] == 0) & (data['speed'].shift(-1) > 0)]
# 显示筛选后的停车点
stopped_points = stops[['vehicle_id', 'time', 'speed']]
print(stopped_points)
# 提取速度为零的时间点，并将其转换为数值类型
zero_speed_times = stops['time'].apply(lambda x: x.timestamp()).values.reshape(-1, 1)
# 计算不同聚类数量下的轮廓系数
silhouette_scores = []
for k in range(2, 40):  # 从 2 开始，因为 1 个簇没有意义
    kmeans = KMeans(n_clusters=k, random_state=0).fit(zero_speed_times)
    labels = kmeans.labels_
    silhouette_avg = silhouette_score(zero_speed_times, labels)
    silhouette_scores.append(silhouette_avg)
# 绘制肘部法则图
plt.figure(figsize=(12, 6))
plt.plot(range(2, 40), silhouette_scores, marker='o')
plt.title('Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('Silhouette Score')
plt.grid(True)
plt.show()
# 根据肘部法则选择聚类数量
optimal_k = np.argmax(silhouette_scores) + 2  # 加 2 是因为我们跳过了 k=1
print(f"Optimal number of clusters: {optimal_k}")
# 获取聚类数为 32 的轮廓系数
silhouette_score_for_k32 = silhouette_scores[31]
print(f"Silhouette Score for k=32: {silhouette_score_for_k32}")
# 使用选择的聚类数量估计周期
kmeans = KMeans(n_clusters=optimal_k, random_state=0).fit(zero_speed_times)
labels = kmeans.labels_
# 绘制每个簇的时间点分布
```

```
plt.figure(figsize=(12, 6))
for i in range(optimal_k):
    plt.plot(zero_speed_times[labels == i], np.zeros_like(zero_speed_times[labels == i]), 'o',
label=f'Cluster {i}')
plt.title('Time Points Distribution')
plt.xlabel('Time')
plt.ylabel('Count')
plt.legend()
plt.show()
import pandas as pd
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt
# 使用 KMeans 算法对时间点进行聚类
kmeans = KMeans(n_clusters=optimal_k, random_state=0).fit(zero_speed_times.reshape(-1, 1))
labels = kmeans.labels_
# 获取每个簇的时间点
cluster_points = {i: zero_speed_times[labels == i] for i in range(optimal_k)}
# 初始化一个列表来存储所有簇对的时间差
time_differences = []
# 计算所有簇对之间的时间差
for i in range(optimal_k):
    for j in range(i+1, optimal_k):
        cluster_i_points = cluster_points[i]
        cluster_j_points = cluster_points[j]
        # 找出簇 i 和簇 j 中最近的时间点
        min_time_diff = np.min(np.abs(cluster_i_points[:, np.newaxis] - cluster_j_points))
        time_differences.append(min_time_diff)
# 对所有时间差进行排序，并获取最小的 31 个
sorted_time_diffs = sorted(time_differences)[:31]
# 绘制最小 31 个时间差的折线图
plt.figure(figsize=(10, 6))
plt.plot(sorted_time_diffs, marker='o')
plt.title('Top 31 Minimum Time Differences Between Clusters (in seconds)')
plt.xlabel('Rank')
plt.ylabel('Time Difference (seconds)')
plt.grid(True)
plt.show()
import numpy as np
# 假设 sorted_time_diffs 是一个已经按时间排序的时间差列表
# sorted_time_diffs = [ ... ]  # 您的时间差列表
def find_longest_stable_segment(time_diffs, max_diff=5):
```

```
    longest_segment = []
    current_segment = []
    for i in range(len(time_diffs) - 1):
        diff = time_diffs[i+1] - time_diffs[i]
        if abs(diff) <= max_diff:
            current_segment.append(time_diffs[i])
        else:
            if len(current_segment) > len(longest_segment):
                longest_segment = current_segment
            current_segment = []
    # 添加最后一个时间差，因为它可能没有被添加到任何段中
    current_segment.append(time_diffs[-1])
    if len(current_segment) > len(longest_segment):
        longest_segment = current_segment
    return longest_segment
# 找到点数最多的时间段
longest_stable_segment = find_longest_stable_segment(sorted_time_diffs, max_diff=3)
# 计算这个时间段的平均时间差并四舍五入到最接近的整数
average_difference_longest_segment = round(np.mean(longest_stable_segment))
print(f"Average difference of the longest stable segment: {average_difference_longest_segment}
seconds")
```

## 附录二　问题二代码（python）

```
#数据预处理
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from statsmodels.tsa.stattools import adfuller
# 读取 CSV 文件
data = pd.read_csv(r"C:\Users\Administrator\Desktop\B 题\附件\附件 2\B1.csv")
# 打印前五行
print(data.head())

#缺失值处理
# 确保时间点是整数
data['time'] = data['time'].astype(int)
# 按车辆 ID 分组，并对每个分组进行处理
grouped = data.groupby('vehicle_id')
# 创建一个空的 DataFrame 来存储处理后的数据
processed_data = pd.DataFrame()
```

```
# 对每个分组进行处理
for group_id, group in grouped:
    # 获取当前分组的时间点列表
    time_points = group['time'].tolist()
    # 生成完整的时间点列表
    full_time_points = list(range(time_points[0], time_points[-1] + 1))
    # 找出缺失的时间点
    missing_time_points = set(full_time_points) - set(time_points)
    # 统计缺失的时间点数量
    missing_count = len(missing_time_points)
    # 打印缺失时间点数量
    print(f"车辆 ID {group_id} 缺失时间点数量: {missing_count}")
    # 对于每个缺失的时间点，创建一个新的行
    for missing_time in missing_time_points:
        new_row = {'time': missing_time, 'vehicle_id': group_id, 'x': 0, 'y': 0}
        processed_data = processed_data.append(new_row, ignore_index=True)
    # 将当前分组的数据添加到处理后的数据中
    processed_data = processed_data.append(group, ignore_index=True)
# 对处理后的数据进行排序
processed_data.sort_values(by=['vehicle_id', 'time'], inplace=True)
# 重置索引
processed_data.reset_index(drop=True, inplace=True)
# 将处理后的数据保存到新的 CSV 文件
processed_data.to_csv(r"C:\Users\Administrator\Desktop\B 题\附件\附件 1\processed_B1.csv",
index=False)

#求取速度
# 读取 CSV 文件
data = pd.read_csv(r"C:\Users\Administrator\Desktop\B 题\附件\附件 2\processed_B1.csv")
# 根据车辆 ID 分组数据
grouped_data = data.groupby('vehicle_id')
# 初始化一个空 DataFrame 来存储结果
velocities = pd.DataFrame(columns=['vehicle_id', 'time', 'instantaneous_speed'])
# 对每个车辆分组进行操作
for vehicle_id, group in grouped_data:
    # 计算 x 和 y 坐标的变化
    group['delta_x'] = group['x'].diff()
    group['delta_y'] = group['y'].diff()
    # 计算时间的变化
    group['delta_t'] = group['time'].diff()
    # 计算瞬时速度
    group['instantaneous_speed'] = ((group['delta_x']**2 + group['delta_y']**2)**0.5) /
```

```
group['delta_t']
    # 移除速度为 NaN 的行
    group = group.dropna(subset=['instantaneous_speed'])
    # 使用.loc[]或.iloc[]来追加结果，而不是.append()
    velocities = velocities.append(group[['vehicle_id', 'time',
'instantaneous_speed']].loc[group.index])
# 显示处理后的速度数据的前几行
print(velocities.head())
# 将处理后的结果导出为 csv 文件
output_file_path = r"C:\Users\Administrator\Desktop\B 题\附件\附件 2\processed1_B2.csv"
velocities.to_csv(output_file_path, index=False)
print(f"处理后的速度数据已导出到 {output_file_path}")

#异常值处理
import pandas as pd
# 假设 'velocities' DataFrame 已经被加载或创建
# 按 'vehicle_id' 分组
grouped_data = velocities.groupby('vehicle_id')
# 定义处理异常值的函数
def handle_abnormal_values(group, name):
    # 计算速度差值
    speed_diff = group['instantaneous_speed'].diff()
    # 识别异常值
    abnormal_indices = speed_diff[abs(speed_diff) > 3].index
    # 初始化异常值列表
    abnormal_values = []
    # 遍历异常值索引，确保索引有效并替换值
    for index in abnormal_indices:
        # 检查索引是否有效
        if index - 1 in group.index and index + 1 in group.index:
            # 将异常值替换为前后时间点的速度均值
            original_speed = group.loc[index, 'instantaneous_speed']
            group.loc[index, 'instantaneous_speed'] = (group.loc[index - 1,
'instantaneous_speed'] +
                                                       group.loc[index + 1,
'instantaneous_speed']) / 2
            # 将原始异常值、替换后的新值、车辆 ID 和时间点添加到列表
            abnormal_values.append((name, index, original_speed, group.loc[index,
'instantaneous_speed']))
    return group, abnormal_values
# 对每个分组独立地进行异常值处理，并收集异常值信息
abnormal_values_list = []
```

```
for name, group in grouped_data:
    cleaned_group, abnormal_values = handle_abnormal_values(group, name)
    abnormal_values_list.extend(abnormal_values)
# 将收集到的异常值信息转换为 DataFrame
abnormal_values_df = pd.DataFrame(abnormal_values_list, columns=['vehicle_id', 'timestamp',
'original_speed', 'new_speed'])
# 打印原始异常值和新值，以及对应的车辆 ID 和时间点
print("原始异常值和新值对比，以及对应的车辆 ID 和时间点：")
print(abnormal_values_df)

#求取红灯时长
# 导入相关库
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
# 读取 CSV 文件
data = pd.read_csv(r"C:\Users\Administrator\Desktop\B 题\附件\附件 2\processed1_B1.csv")
# 打印前五行
print(data.head())
import pandas as pd
import matplotlib.pyplot as plt
# 读取 CSV 文件
data = pd.read_csv(r"C:\Users\Administrator\Desktop\B 题\附件\附件 2\processed1_B1.csv")
# 按车辆 ID 分组数据
grouped_data = data.groupby('vehicle_id')
# 随机选择五辆车
random_vehicles = grouped_data.apply(lambda x: x.sample(1)).reset_index(drop=True)
# 绘制图像
for index, row in random_vehicles.iterrows():
    vehicle_id = row['vehicle_id']
    group = data[data['vehicle_id'] == vehicle_id]
    plt.figure(figsize=(10, 6))  # 创建一个新的图形窗口
    plt.plot(group['time'], group['instantaneous_speed'], label=f'Vehicle {vehicle_id}')
    plt.title(f'Vehicle {vehicle_id} Instantaneous_speed Over Time')
    plt.xlabel('Time')
    plt.ylabel('Instantaneous_speed')
    plt.legend()
plt.show()

import pandas as pd
# 读取 CSV 文件
data = pd.read_csv(r"C:\Users\Administrator\Desktop\B 题\附件\附件 2\processed1_B1.csv")
```

```
# 初始化一个空 DataFrame 来存储结果
df1 = pd.DataFrame(columns=['vehicle_id', 'interval-red'])
# 按车辆 ID 分组数据
grouped_data = data.groupby('vehicle_id')
# 筛选出同一辆车的记录
for vehicle_id, group in grouped_data:
    # 筛选出速度为 0 的点
    zero_speed_points = group[group['instantaneous_speed'] == 0]
    # 检查是否有不少于五个时间点的速度为 0
    if zero_speed_points.shape[0] >= 5:
        # 计算最后一个速度为零的时间点的时间和第一个速度为零的点的时间差
        last_zero_speed_time = zero_speed_points['time'].iloc[-1]
        first_zero_speed_time = zero_speed_points['time'].iloc[0]
        interval_red = last_zero_speed_time - first_zero_speed_time
        # 将结果添加到 df DataFrame
        df1 = df1.append({'vehicle_id': vehicle_id, 'interval-red': interval_red},
ignore_index=True)
    else:
        # 如果少于 5 个时间点，则 interval-red 记为 0
        df1 = df1.append({'vehicle_id': vehicle_id, 'interval-red': 0}, ignore_index=True)
# 显示结果
print(df1)
# 确保将以下路径替换为您实际想要保存的路径
output_file_path = r"C:\Users\Administrator\Desktop\B 题\附件\附件 2\processed2_B1.csv"
# 将结果保存到新的 CSV 文件
df1.to_csv(output_file_path, index=False)

# 找出 interval-red 最大值的车辆 ID 以及结果
max_interval_red = df1['interval-red'].max()
max_vehicle_id = df1[df1['interval-red'] == max_interval_red]['vehicle_id'].iloc[0]
max_vehicle_id, max_interval_red
print("红灯时长为",max_interval_red)

#求取总周期
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
import matplotlib.pyplot as plt
import numpy as np
# 读取 CSV 文件
data = pd.read_csv(r"C:\Users\Administrator\Desktop\B 题\附件\附件 2\processed1_B1.csv")
# 检查 CSV 文件中的列名
```

```
print(data.columns)
# 将时间转换为时间戳
data['time'] = pd.to_datetime(data['time'], unit='s')  # 假设时间是以秒为单位
# 找出速度从非零变为零的点
stops = data[(data['instantaneous_speed'] == 0) & (data['instantaneous_speed'].shift(-1) > 0)]
# 显示筛选后的停车点
stopped_points = stops[['vehicle_id', 'time', 'instantaneous_speed']]
print(stopped_points)
# 提取速度为零的时间点，并将其转换为数值类型
zero_speed_times = stops['time'].apply(lambda x: x.timestamp()).values.reshape(-1, 1)

# 计算不同聚类数量下的轮廓系数
silhouette_scores = []
for k in range(2, 10):  # 从 2 开始，因为 1 个簇没有意义
    kmeans = KMeans(n_clusters=k, random_state=0).fit(zero_speed_times)
    labels = kmeans.labels_
    silhouette_avg = silhouette_score(zero_speed_times, labels)
    silhouette_scores.append(silhouette_avg)
# 绘制肘部法则图
plt.figure(figsize=(12, 6))
plt.plot(range(2, 10), silhouette_scores, marker='o')
plt.title('Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('Silhouette Score')
plt.grid(True)
plt.show()
# 根据肘部法则选择聚类数量
optimal_k = np.argmax(silhouette_scores) + 2  # 加 2 是因为我们跳过了 k=1
print(f"Optimal number of clusters: {optimal_k}")

# 获取聚类数为 25 的轮廓系数
silhouette_score_for_k25 = silhouette_scores[24]
print(f"Silhouette Score for k=5: {silhouette_score_for_k25}")

import pandas as pd
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt
# 使用 KMeans 算法对时间点进行聚类
kmeans = KMeans(n_clusters=optimal_k, random_state=0).fit(zero_speed_times.reshape(-1, 1))
labels = kmeans.labels_
# 获取每个簇的时间点
```

```
cluster_points = {i: zero_speed_times[labels == i] for i in range(optimal_k)}
# 初始化一个列表来存储所有簇对的时间差
time_differences = []
# 计算所有簇对之间的时间差
for i in range(optimal_k):
    for j in range(i+1, optimal_k):
        cluster_i_points = cluster_points[i]
        cluster_j_points = cluster_points[j]
        # 找出簇 i 和簇 j 中最近的时间点
        min_time_diff = np.min(np.abs(cluster_i_points[:, np.newaxis] - cluster_j_points))
        time_differences.append(min_time_diff)
# 对所有时间差进行排序，并获取最小的 25 个
sorted_time_diffs = sorted(time_differences)[:25]
# 绘制最小 25 个时间差的折线图
plt.figure(figsize=(10, 6))
plt.plot(sorted_time_diffs, marker='o')
plt.title('Top 25 Minimum Time Differences Between Clusters (in seconds)')
plt.xlabel('Rank')
plt.ylabel('Time Difference (seconds)')
plt.grid(True)
plt.show()

import numpy as np
# 假设 sorted_time_diffs 是一个已经按时间排序的时间差列表
# sorted_time_diffs = [ ... ]  # 您的时间差列表
def find_longest_stable_segment(time_diffs, max_diff=5):
    longest_segment = []
    current_segment = []
    for i in range(len(time_diffs) - 1):
        diff = time_diffs[i+1] - time_diffs[i]
        if abs(diff) <= max_diff:
            current_segment.append(time_diffs[i])
        else:
            if len(current_segment) > len(longest_segment):
                longest_segment = current_segment
            current_segment = []
    # 添加最后一个时间差，因为它可能没有被添加到任何段中
    current_segment.append(time_diffs[-1])
    if len(current_segment) > len(longest_segment):
        longest_segment = current_segment
    return longest_segment
# 找到点数最多的时间段
```

```
longest_stable_segment = find_longest_stable_segment(sorted_time_diffs, max_diff=3)
# 计算这个时间段的平均时间差并四舍五入到最接近的整数
average_difference_longest_segment = round(np.mean(longest_stable_segment))
print(f"Average difference of the longest stable segment: {average_difference_longest_segment} seconds")

#补充
#当聚类数目较多，但最终确定的聚类个数较少且结果不符合实际情况，根据肘部法则图以及轮廓系数重新确定聚类数目，后续代码如下：
# 获取聚类数为 14 的轮廓系数
silhouette_score_for_k5 = silhouette_scores[4]
print(f"Silhouette Score for k=5: {silhouette_score_for_k5}")

# 使用选择的聚类数量估计周期
kmeans = KMeans(n_clusters=14, random_state=0).fit(zero_speed_times)
labels = kmeans.labels_
# 绘制每个簇的时间点分布
plt.figure(figsize=(12, 6))
for i in range(14):
    plt.plot(zero_speed_times[labels == i], np.zeros_like(zero_speed_times[labels == i]), 'o', label=f'Cluster {i}')
plt.title('Time Points Distribution')
plt.xlabel('Time')
plt.ylabel('Count')
plt.legend()
plt.show()

# 设置聚类数量
optimal_k = 14
# 使用 KMeans 进行聚类
kmeans = KMeans(n_clusters=optimal_k, random_state=0).fit(zero_speed_times.reshape(-1, 1))
labels = kmeans.labels_
# 获取每个簇的时间点
cluster_points = {i: zero_speed_times[labels == i] for i in range(optimal_k)}
# 初始化一个列表来存储所有簇对的时间差
time_differences = []
# 计算所有簇对之间的时间差
for i in range(optimal_k):
    for j in range(i+1, optimal_k):
        cluster_i_points = cluster_points[i]
        cluster_j_points = cluster_points[j]
        # 找出簇 i 和簇 j 中最近的时间点
```

```
            min_time_diff = np.min(np.abs(cluster_i_points[:, np.newaxis] - cluster_j_points))
            time_differences.append(min_time_diff)
# 对所有时间差进行排序，并获取最小的 13 个
sorted_time_diffs = sorted(time_differences)[:14]
# 绘制最小 13 个时间差的折线图
plt.figure(figsize=(10, 6))
plt.plot(sorted_time_diffs, marker='o')
plt.title('Top 13 Minimum Time Differences Between Clusters (in seconds)')
plt.xlabel('Rank')
plt.ylabel('Time Difference (seconds)')
plt.grid(True)
plt.show()
# 返回绘制的图表
plt.show()

import numpy as np
# 假设 sorted_time_diffs 是一个已经按时间排序的时间差列表
# sorted_time_diffs = [ ... ]  # 您的时间差列表
def find_longest_stable_segment(time_diffs, max_diff=5):
    longest_segment = []
    current_segment = []
    for i in range(len(time_diffs) - 1):
        diff = time_diffs[i+1] - time_diffs[i]
        if abs(diff) <= max_diff:
            current_segment.append(time_diffs[i])
        else:
            if len(current_segment) > len(longest_segment):
                longest_segment = current_segment
            current_segment = []
    # 添加最后一个时间差，因为它可能没有被添加到任何段中
    current_segment.append(time_diffs[-1])
    if len(current_segment) > len(longest_segment):
        longest_segment = current_segment
    return longest_segment
# 找到点数最多的时间段
longest_stable_segment = find_longest_stable_segment(sorted_time_diffs, max_diff=3)
# 计算这个时间段的平均时间差并四舍五入到最接近的整数
average_difference_longest_segment = round(np.mean(longest_stable_segment))
print(f"Average difference of the longest stable segment: {average_difference_longest_segment}
seconds")

#当聚类数目本身较小，聚类的数目也少，计算时间点差值得到周期的 k 倍（k≠1）时，在
```

原始代码最后补充：

```
# 重新计算相邻元素之间的差值并排序
differences_between_pairs = [sorted_time_diffs[i+1] - sorted_time_diffs[i] for i in range(len(sorted_time_diffs)-1)]
sorted_differences_between_pairs = sorted(differences_between_pairs)
# 计算差值相隔不超过五的均值
filtered_differences = [diff for diff in sorted_differences_between_pairs if diff <= 5]
mean_difference = np.mean(filtered_differences) if filtered_differences else 0
# 打印排序后的差值和差值相隔不超过五的均值
sorted_differences_between_pairs, mean_difference
#车辆比例
import pandas as pd
import numpy as np
import random
# 读取 CSV 文件
data = pd.read_csv(r"C:\Users\Administrator\Desktop\B 题\附件\附件 2\processed1_B1.csv")
# 初始化一个空 DataFrame 来存储结果
df1 = pd.DataFrame(columns=['vehicle_id', 'interval-red'])
# 按车辆 ID 分组数据
grouped_data = data.groupby('vehicle_id')
# 获取所有唯一的车辆 ID
vehicle_ids = list(grouped_data.groups.keys())
# 计算需要选择的车辆数量（80%）
num_vehicles_to_select = int(len(vehicle_ids) * 0.8)
# 随机选择 80%的车辆 ID
selected_vehicle_ids = random.sample(vehicle_ids, num_vehicles_to_select)
# 筛选出选中的车辆 ID 的数据
selected_data = data[data['vehicle_id'].isin(selected_vehicle_ids)]
# 将结果保存到新的 CSV 文件
output_file_path = r"C:\Users\Administrator\Desktop\B 题\附件\其他\B1 随机数据.csv"
selected_data.to_csv(output_file_path, index=False)
# 只对选中的车辆 ID 进行后续的数据处理
for vehicle_id, group in grouped_data:
    if vehicle_id in selected_vehicle_ids:
        zero_speed_points = group[group['instantaneous_speed'] == 0]
        if zero_speed_points.shape[0] >= 5:
            last_zero_speed_time = zero_speed_points['time'].iloc[-1]
            first_zero_speed_time = zero_speed_points['time'].iloc[0]
            interval_red = last_zero_speed_time - first_zero_speed_time
            df1 = df1.append({'vehicle_id': vehicle_id, 'interval-red': interval_red}, ignore_index=True)
        else:
```

```
                df1 = df1.append({'vehicle_id': vehicle_id, 'interval-red': 0}, ignore_index=True)
# 显示结果
print(df1)
```

注：除此部分外，剩下代码同“问题二代码”

```
#定位误差
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from statsmodels.tsa.stattools import adfuller
# 读取 CSV 文件
data = pd.read_csv(r"C:\Users\Administrator\Desktop\B 题\附件\附件 2\B1.csv")
# 显示数据框的前几行以了解其结构
data.head()
# 计算每辆车在每一点移动的距离
data['distance'] = (data.groupby('vehicle_id')['x'].diff()**2 +
data.groupby('vehicle_id')['y'].diff()**2).apply(lambda x: x**0.5)
# 由于没有前一个数据点进行距离计算，将每辆车数据的第一行填充为 NaN
data['distance'].fillna(value=pd.NA, inplace=True)
# 显示更新后的数据框的前几行
data.head()

# 定义判断车辆是否停止的阈值
# 以 1 米/秒作为阈值
stopped_threshold = 1.0
# 标记车辆被认为停止的时间点
data['stopped'] = data['distance'] < stopped_threshold
# 按车辆 ID 分组数据，找出连续的停止序列
stopped_sequences = data[data['stopped']].groupby('vehicle_id')['time'].agg([lambda x: list(x)])
stopped_sequences.head()

# 提取每个车辆停止序列的持续时间
stopped_durations = [len(seq) for seq_list in stopped_sequences.values for seq in seq_list]
# 计算停止持续时间的频率分布
duration_freq = pd.Series(stopped_durations).value_counts().sort_index()
duration_freq.head()

# 定义位置误差的标准差
error_std_devs = [1.0, 2.0, 3.0]  # 1 米、2 米和 3 米标准差
# 模拟具有不同定位误差的数据
```

```
simulated_data_with_errors = []
for std_dev in error_std_devs:
    # 向 X 和 Y 坐标添加随机噪声
    data['x_noisy'] = data['x'] + np.random.normal(0, std_dev, len(data))
    data['y_noisy'] = data['y'] + np.random.normal(0, std_dev, len(data))
    # 对模拟数据应用相同的模型（例如，距离计算）
    data['distance_noisy'] = (data.groupby('vehicle_id')['x_noisy'].diff()**2 + data.groupby('vehicle_id')['y_noisy'].diff()**2).apply(lambda x: x**0.5)
    data['distance_noisy'].fillna(value=pd.NA, inplace=True)
    data['stopped_noisy'] = data['distance_noisy'] < stopped_threshold
    stopped_sequences_noisy = data[data['stopped_noisy']].groupby('vehicle_id')['time'].agg([lambda x: list(x)])
    stopped_durations_noisy = [len(seq) for seq_list in stopped_sequences_noisy.values for seq in seq_list]
    duration_freq_noisy = pd.Series(stopped_durations_noisy).value_counts().sort_index()
    simulated_data_with_errors.append(duration_freq_noisy)
simulated_data_with_errors

import matplotlib.pyplot as plt
# 创建子图用于每个标准差的频率分布图
fig, axes = plt.subplots(1, len(error_std_devs), figsize=(15, 5))
# 为每个标准差绘制频率分布图
for i, std_dev in enumerate(error_std_devs):
    axes[i].bar(simulated_data_with_errors[i].index, simulated_data_with_errors[i], color='blue')
    axes[i].set_title(f'position error SD: {std_dev} /m') #定位误差标准差
    axes[i].set_xlabel('Length of stay period /s') #停留时间段长度
axes[i].set_ylabel('rate') # 频率
# 调整布局
plt.tight_layout()
# 显示图表
plt.show()

#A 车流量
import numpy as np
# 随机选择车辆 ID 子集以模拟不同的车流量水平
# 例如，可以保持原始数据的基本结构不变，但改变车辆 ID 的数量来模拟不同的车流量水平
vehicle_ids = data['vehicle_id'].unique()
subset_sizes = [0.5, 0.75, 1.0]  # 50%, 75%, 和 100% 的原始数据
# 为不同的车流量水平模拟数据
simulated_data = []
```

```
for size in subset_sizes:
    selected_ids = np.random.choice(vehicle_ids, int(len(vehicle_ids) * size), replace=False)
    subset_data = data[data['vehicle_id'].isin(selected_ids)]
    simulated_data.append(subset_data)
# 在模拟数据上应用相同的模型（例如，用于估计信号灯周期的模型）
simulated_results = []
for subset in simulated_data:
    subset['distance'] = (subset.groupby('vehicle_id')['x'].diff()**2 + subset.groupby('vehicle_id')['y'].diff()**2).apply(lambda x: x**0.5)
    subset['distance'].fillna(value=pd.NA, inplace=True)
    subset['stopped'] = subset['distance'] < stopped_threshold
    stopped_sequences_subset = subset[subset['stopped']].groupby('vehicle_id')['time'].agg([lambda x: list(x)])
    stopped_durations_subset = [len(seq) for seq_list in stopped_sequences_subset.values for seq in seq_list]
    duration_freq_subset = pd.Series(stopped_durations_subset).value_counts().sort_index()
    simulated_results.append(duration_freq_subset)
simulated_results

# 绘制不同车流量水平下的信号灯周期估计散点图
import matplotlib.pyplot as plt
# 创建散点图以展示不同车流量水平下的频率分布
fig, ax = plt.subplots(figsize=(15, 5))
# 为每个车流量水平绘制散点图
for i, size in enumerate(subset_sizes):
    ax.scatter(simulated_results[i].index, simulated_results[i], label=f'车流量水平: {size*100}%')
# 设置图表标题和标签
ax.set_title('Signal light cycle estimation') #不同车流量水平下的信号灯周期估计
ax.set_xlabel('Length of stay period /s') #停留时间段长度
ax.set_ylabel('rate') #频率
ax.legend()
# 调整布局
plt.tight_layout()
# 显示图表
plt.show()
```

## 附录三　问题三代码（python）

问题三前部分代码也同问题二代码，剩余如下：

```
# .....重复略
import pandas as pd
```

```
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt
# 使用 KMeans 算法对时间点进行聚类
kmeans = KMeans(n_clusters=optimal_k, random_state=0).fit(zero_speed_times.reshape(-1, 1))
labels = kmeans.labels_
# 获取每个簇的时间点
cluster_points = {i: zero_speed_times[labels == i] for i in range(optimal_k)}
# 初始化一个列表来存储所有簇对的时间差
time_differences = []
# 计算所有簇对之间的时间差
for i in range(optimal_k):
    for j in range(i+1, optimal_k):
        cluster_i_points = cluster_points[i]
        cluster_j_points = cluster_points[j]
        # 找出簇 i 和簇 j 中最近的时间点
        min_time_diff = np.min(np.abs(cluster_i_points[:, np.newaxis] - cluster_j_points))
        time_differences.append(min_time_diff)
# 对所有时间差进行排序，并获取最小的 n² 个
sorted_time_diffs = sorted(time_differences)[:n²] #簇个数的平方
# 绘制最小 n² 个时间差的折线图
plt.figure(figsize=(10, 6))
plt.plot(sorted_time_diffs, marker='o')
plt.title('Top 160 Minimum Time Differences Between Clusters (in seconds)')
plt.xlabel('Rank')
plt.ylabel('Time Difference (seconds)')
plt.grid(True)
plt.show()

import numpy as np
def find_stable_segments(time_diffs, max_diff, min_length=10):
    segments = []
    current_segment = []
    for i in range(len(time_diffs) - 1):
        diff = time_diffs[i+1] - time_diffs[i]
        if abs(diff) <= max_diff:
            current_segment.append(time_diffs[i])
        else:
            if current_segment and len(current_segment) >= min_length:
                segments.append(current_segment)
            current_segment = []
    # 添加最后一个时间差，因为它可能没有被添加到任何段中
```

```
        if current_segment and len(current_segment) >= min_length:
            segments.append(current_segment)
        return segments
def calculate_average_differences(segments):
        averages = []
        for segment in segments:
            average_diff = round(np.mean(segment))  # 四舍五入到最接近的整数
            averages.append(average_diff)
        return averages
# 找到稳定时间段
segments = find_stable_segments(sorted_time_diffs, max_diff=3, min_length=10)
# 计算各段的平均时间差
average_differences = calculate_average_differences(segments)
# 打印结果
print(average_differences)

#求 k 倍周期
new_average_differences = [average_differences[i] - average_differences[i-1]
for i in range(1, len(average_differences))]
# 打印新的列表
print(new_average_differences)
```

经计算，C3 存在周期切换时刻，故继续处理数据代码如下：

```
max_diff = 3
# 假设我们要找的特定值是 88
target_value = 88

# 找到特定值在 new_average_differences 中的索引
target_index = new_average_differences.index(target_value)

# 由于 new_average_differences 中的值是由 average_differences 的相邻元素相减得到的，
# 所以我们可以通过 target_index 来找到 average_differences 中对应的两个时间差
prev_time_diff = average_differences[target_index]
current_time_diff = average_differences[target_index + 1]

# 现在我们需要找到这两个时间差在原始数据 segments 中的位置
# 我们将遍历 segments 并计算每个段的时间差，然后与 prev_time_diff 和 current_time_diff
进行比较

# 存储找到的簇中心时间点
cluster_centers = []
```

```
# 遍历 segments 中的每个段
for segment in segments:
    segment_mean = np.mean(segment)
    # 如果段的时间差接近 prev_time_diff 或 current_time_diff，则记录该段的时间点
    if abs(segment_mean - prev_time_diff) <= max_diff or abs(segment_mean - current_time_diff) <= max_diff:
        # 假设我们想要段中的第一个时间点作为簇中心的时间点
        cluster_center = segment[0]
        cluster_centers.append(cluster_center)

# 打印找到的簇中心时间点
print(f"簇中心时间点对应于差值{target_value}的范围:")
print(cluster_centers)
```

## 附录四　问题四代码（使用Kaggle）

```
import pandas as pd
import numpy as np
from scipy.stats import entropy
# 读取数据
data = pd.read_csv('/kaggle/input/d111111/D.csv')  # 数据集上传到 Kaggle
# 查看数据的前几行以了解其结构
print(data.head())

# 计算每辆车的速度
data['speed'] = ((data['x'] - data['x'].shift(1))**2 + (data['y'] - data['y'].shift(1))**2)**0.5
# 过滤出速度较小的点，这些点可能表示车辆接近路口
slow_points = data[data['speed'] < 1]
# 查看速度较慢的数据点
print(slow_points.head())

# 距离阈值法
intersection_point = data[['x', 'y']].head(5).mean()
# 计算每辆车到路口的距离
data['distance_to_intersection'] = ((data['x'] - intersection_point['x'])**2 + (data['y'] - intersection_point['y'])**2)**0.5
# 设置一个距离阈值，以确定车辆是否接近路口
distance_threshold = 10
close_to_intersection = data[data['distance_to_intersection'] < distance_threshold]
# 查看接近路口的数据点
print(close_to_intersection.head())
```

```
# 计算车辆的位置变化
data['delta_x'] = data['x'].diff()
data['delta_y'] = data['y'].diff()
# 计算位置变化的绝对值
data['abs_delta_x'] = data['delta_x'].abs()
data['abs_delta_y'] = data['delta_y'].abs()
# 筛选出位置变化较大的点，这些点可能表示车辆正在接近或离开路口
significant_movement = data[(data['abs_delta_x'] > 5) | (data['abs_delta_y'] > 5)]
# 查看位置变化较大的数据点
print(significant_movement.head())

# 为每辆车确定到达和离开路口的时间
arrival_departure_times = significant_movement.groupby('vehicle_id')['time'].agg(['min', 'max'])
arrival_departure_times = arrival_departure_times.reset_index()
# 查看车辆的到达和离开时间
print(arrival_departure_times.head())
# 设置信号灯周期的可能范围
min_cycle = 86
max_cycle = 115
num_simulations = 1000
np.random.seed(0)
random_cycles = np.random.randint(min_cycle, max_cycle + 1, num_simulations)
# 计算整个数据集的到达时间间隔
arrival_intervals_all = data.groupby('vehicle_id')['time'].diff().dropna()
# 计算数据集到达时间间隔的熵作为匹配程度得分
matching_scores_entropy_all = []
for cycle in random_cycles:
    intervals_mod_cycle = arrival_intervals_all % cycle
    distribution, _ = np.histogram(intervals_mod_cycle, bins=np.arange(0, cycle + 1, 1))
    distribution = distribution / np.sum(distribution)
    score = entropy(distribution)
    matching_scores_entropy_all.append(score)
# 将熵值与周期对应起来
matching_results_entropy_all = pd.DataFrame({'cycle': random_cycles,
'matching_score_entropy_all': matching_scores_entropy_all})
# 找出熵值最低的周期，即分布最有序的周期
best_cycle_entropy_all =
matching_results_entropy_all.sort_values(by='matching_score_entropy_all').iloc[0]
# 输出最佳周期
print("Best cycle (entropy method, all data):", best_cycle_entropy_all['cycle'])
# 保存结果
```

```
best_cycle_entropy_all.to_csv('/kaggle/working/best_cycle_entropy_all.csv', index=False)
```