中国研究生创新实践系列大赛

“华为杯”第二十一届中国研究生数学建模竞赛

题　目： 多模型融合的高速公路应急车道启用决策与效果评估

摘　要：

随着城市化进程的加快和机动车数量的不断增加，交通拥堵已成为主要城市问题之一，为应对这一挑战，许多研究开始探索非紧急情况下临时开放应急车道以缓解交通压力的可能性。本研究基于节假日高速公路的实时交通数据，构建数学模型发掘高速公路特定路段即将发生拥堵的条件，评估临时借用应急车道对缓解路段拥堵的作用，为应急车道的启用决策提供理论依据。

针对第一问，利用预处理数据，可视化交通流参数的变化规律，建立时间序列与 LSTM 的交通流拥堵模型预测道路交通状态，并利用预处理数据验证模型的有效性。使用 YOLO 与目标追踪算法预处理视频数据提取交通参数，分析交通流随时间变化的规律；以 2min 为采样单位再次处理视频数据，并建立传统交通拥堵指数模型，获得带有拥挤度的数据集与拥堵参考阈值，采用 LSTM 与时间序列模型构建交通流拥堵预测模型，并使用粒子群算法自适应更新交通拥堵的参考阈值，对第三个检测点的数据进行预测，划分交通拥堵状态。利用另外三个检测点的数据，划分数据集与验证集，从准确率、召回率、调和均值、阈值验证模型的有效性。

针对第二问，利用分类算法构建高速公路应急车道模型，为决策者提供启用应急车道的理论依据。首先通过当前时间间隔内的车流流速和交通量来构建交通流中断概率模型，预测下一时间间隔内交通流出现中断的概率，从而测定该道路下开启应急车道的速度、流量、密度阈值；通过绘制各个拥堵等级在不同检测点的时间分布图，得到拥堵等级的阈值。利用实时交通流量数据和预测模型来构建决策模型，从而根据阈值判断是否启用应急车道为决策者提供理论依据。

针对第三问，利用多种机器学习分类算法进行结果对比，寻找适合当前交通状况的最优的应急车道启用模型，利用消散时长、车辆数、长度对比，量化模型的作用。结合已有的交通流拥堵预测模型，预测未来交通参数，达到实时决策。利用常见的六种机器学习分类算法搭建模型，进行对比分析，并使用车流波理论，通过计算启用前后车道的消散时间、消散长度、消散车辆数量化应急车道启用模型的作用。

针对第四问，采用基于遗传算法对成本以及交通信息熵进行优化，得出基于最优监控点布局。首先，结合 C 点到 D 点的道路长度与监控器的识别范围，确定需要监控的路段和设备覆盖范围。其次，通过历史交通流参数及 LSTM 预测模型，获取待选监控点处的交通参数。二者线性加权，在满足降低成本和提升启用应急车道科学性的双优化目标下，通过遗传算法进行视频监控点布置优化，得到最优的监控点布置方案。

关键词： 应急车道、LSTM、时间序列分析、粒子群算法、交通流中断概率、支持向量、神经网络、目标优化、遗传算法

目录

一、 问题背景与重述

1.1 问题背景

随着城市化进程的加速和机动车数量的急剧增长，交通拥堵已成为全球各大城市的主要问题之一，特别是在高峰时段，交通流量激增导致的拥堵极大影响了人们的出行效率，并带来了环境污染、能源消耗等诸多负面效应。如何有效缓解交通拥堵、提高道路的通行能力，已经成为交通管理者和学术研究者重点关注的课题。

在高速公路和城市快速路的交通管理中，应急车道作为关键组成部分，主要用于紧急救援和事故处理，以保障特种车辆的通行顺畅。然而，通常情况下，应急车道处于闲置状态，未被用于常规交通管理中。近年来，随着交通压力的增大，研究人员开始探索临时开放应急车道来缓解拥堵的可行性。国内外的研究表明，在某些特定条件下，适度、合理地临时启用应急车道，能够有效缓解道路交通压力，提升整体通行能力。

国际上，如 INRIX Index 等交通拥堵评价指标系统，基于路段速度、交通密度、流量等因素，已经在许多国家用于实时分析和发布交通拥堵信息。欧美国家早在 21 世纪初就开始尝试通过应急车道的开放来缓解高速公路的交通压力，但大多数采用的是固定时段的静态管理方式，未能根据实时交通情况灵活调整。相比之下，现代高速公路的交通瓶颈具有突发性和短时性，传统的静态控制模式难以充分发挥应急车道在动态交通环境中的效用。

本研究旨在基于高速公路上的实时交通数据，探讨在非紧急情况下临时开放应急车道的可行性和管理策略。我们将通过构建数学模型，分析不同观测点的交通流量、密度、速度等参数，预测拥堵发生的时间和地点，制定合理的应急车道启用策略。同时，研究还将讨论如何在紧急救援时快速恢复应急车道的畅通，确保不影响救援行动的顺利进行。通过结合国内外的研究与实践经验，本研究将为决策者提供科学的理论依据，提出应急车道开放管理的优化方案，既能有效缓解交通拥堵，又能确保道路安全和救援效率。

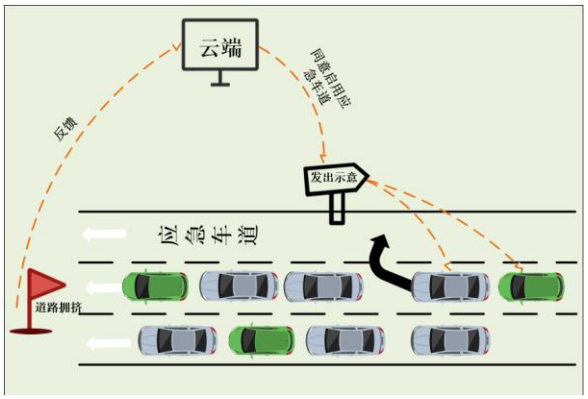

图 1-1　应急车道启用示意图

1.2 问题重述

基于前述背景，该题目提供了四个点位的交通监控视频附件，基于四个附件内容，拟要解决的研究问题：

1. 通过对四个观测点的交通流数据分析：

- 统计交通流参数（车流密度，流量，速度等）随时间变化的规律，同时确保统计

足够精细，以便未来构建准确的模型。

- 基于观测点的基本交通参数（车流密度、流量、速度等）和路段的情况，建立交通流拥堵预测模型，给出第三观测点至第四观测点之间可能出现持续拥堵状态的实时预警，预警时间至少为拥堵发生前 10 分钟，并提供预警依据。
- 利用视频监控数据验证所建立的交通流拥堵预测模型的有效性。

2. 应急车道启用模型构建：构建合理的应急车道启用模型，为决策者提供科学依据，确保在特定情况下临时启用应急车道能够有效缓解交通拥堵。
3. 实时启用应急车道的决策算法设计： 基于交通监控数据，设计合理的规则或算法，实时决策是否启用应急车道，并对应急车道启用的效果进行量化评估。
4. 监控点优化： 现有监控系统并未考虑应急车道临时启用的需求。为了提升第三观测点至第四观测点之间路段应急车道临时启用决策的科学性，并控制成本，研究应如何布置视频监控点？并说明布置理由。

二、 基本假设和符号说明

2.1 基本假设

1. 假设在建模过程中，不考虑交通事故对车流的影响，所有交通流量变化仅由车辆聚集、路段瓶颈和车道数量限制等非事故因素引起。
2. 单向通行假设：假设研究的高速公路段是单向通行的，即车辆只会从起点驶向终点，不考虑逆向行驶的情况。
3. 假设提供的视频数据完整无缺，并且没有缺帧、丢帧或因天气、设备故障等导致的图像质量下降问题。
4. 空间一致性假设：假设四个观测点的空间位置是固定且已知的，不会由于数据采集设备的移动或误差导致监控区域的偏移。

2.2 符号说明

表 2-1 符号说明

符号	说明	单位
L_{car}	每辆车的平均长度	m
d_{safe}	车辆之间的安全间隔	m
L_{road}	估算道路长度	m
q_k	道路 k 的流量	vel/h
C	基准交通量	vel/h
t_k	实际行程时间	s
l_k	道路 k 的长度	m
v_{dk}	期望速度	km/h
q	定义交通流量	c
L	路段长度	m
Q_{max}	流量阈值	vel/h
K	车流密度	vel/km
TPI	交通拥堵指数	
DC	交通拥堵指标	
ω	权重系数	

三、 技术路线图

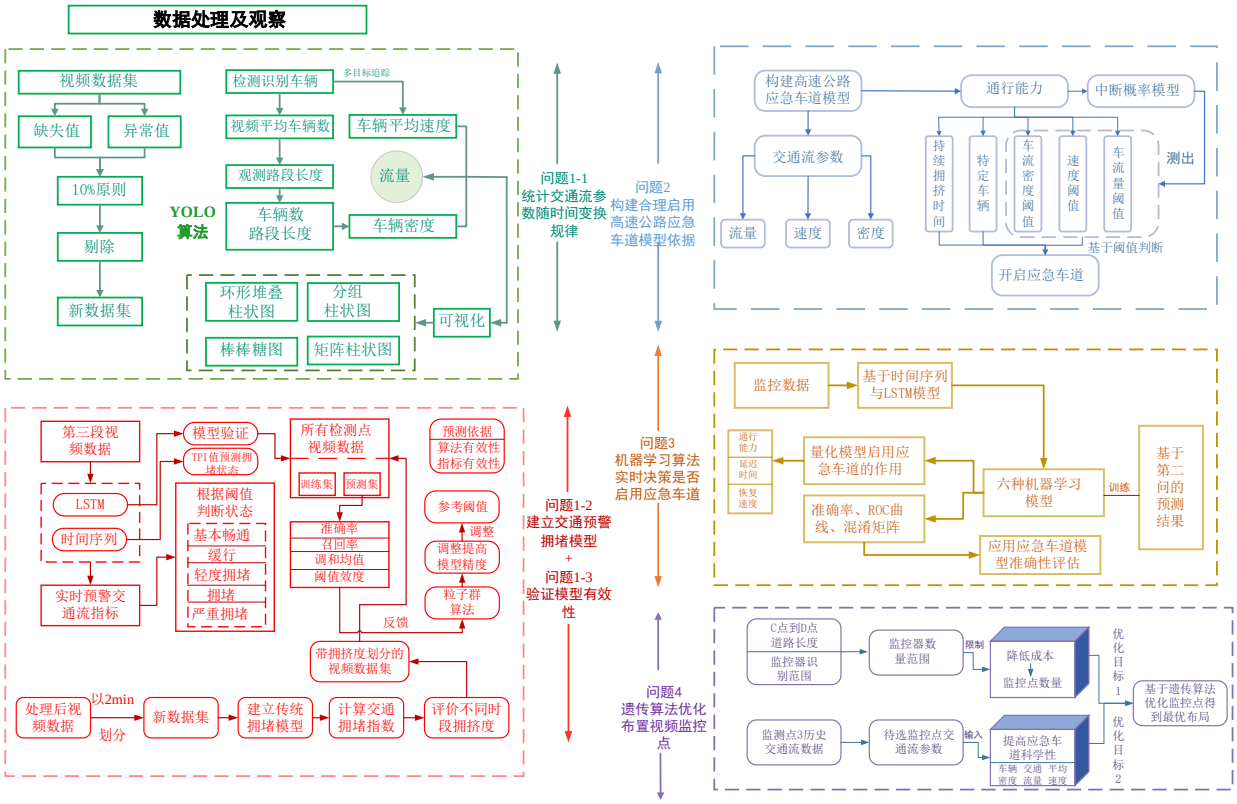

图 3-1　总流程图

四、 问题一

4.1 问题分析

根据题目要求需要得到交通流参数，而数据集是视频的形式，所以考虑图像识别与多目标跟踪提取视频中的交通参数。对于问题一统计四个观测点的基本数据随时间的变化，我们首先利用算法以每 10min 采样得到数据集，然后分别对四个观测点进行可视化分析。对于第二问需要建立交通流拥堵模型，需要更加精细化的数据，我们对数据集再次以 2min 采样频率得到数据集，并且参考相关的文献，首先建立传统的交通拥挤模型，得到初始的拥堵阈值；再结合时间序列分析和 LSTM 算法构建交通流拥堵模型，并且利用粒子群算法自适应更新阈值以及设计报警函数。第三问利用已有的数据集对构建的拥堵模型验证。

4.2 数据预处理

首先对于在检测点 1 在 5 月 1 日 12：57 左右的视频进行处理处理的思路流程图如图 4-1 所示。

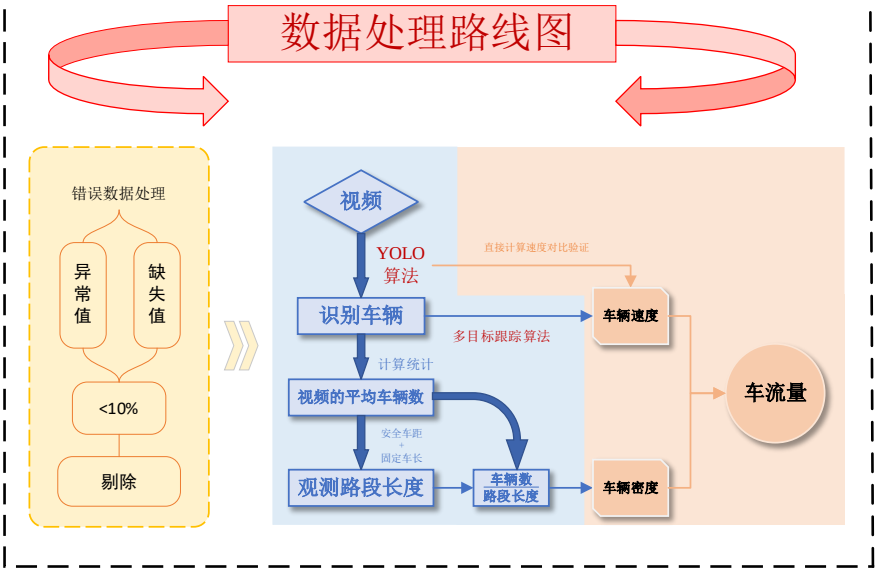

图 4-1 数据处理路线图

视频时间流率的计算：本文发现视频中时间与现实中的时间流速不同，可以换算为 1：10，具体换算为实际时间如表 4-1 所示。

表 4-1 视频时长与实际时长

检测点	第一个视频时长/实际时长	第二个视频时长/实际时长	第三个视频时长/实际时长
检测点 1	7min7s / 1h11min10s	7min11s / 1h11min50s	
检测点 2	7min10s / 1h11min40s	7min10s / 1h11min40s	7min14s / 1h12min20s
检测点 3	13min47s / 2h17min50s	13min48s / 2h18min0s	
检测点 4	13min47s / 2h17min50s	13min47s / 2h17min50s	

视频帧处理与车辆识别：在确定具体的时间流率比后，首先我们使用 OpenCV(Open Source Computer Vision Library)加载视频，并获取视频的帧数和帧率信息。为了从视频帧中识别出车辆及其类型，我们采用了 YOLO(YouOnlyLookOnce)深度学习目标检测算法[1]。

在处理视频时，我们对每一帧进行如下处理：

•帧提取与处理：首先，使用 OpenCV 提取视频的每一帧。

•目标检测：对于每一帧，使用 YOLO 算法进行目标检测。YOLO 会返回多个检测结果，每个结果包括检测框的位置及其对应的类别标签。

•车辆识别与过滤：我们通过过滤 YOLO 的检测结果，仅保留与车辆相关的类别，如小轿车、卡车等，并统计每帧中检测到的车辆数量。

•缺失值处理：如果某一帧中未能检测到车辆（即车辆数量为 0），我们使用前后两帧检测到的车辆数量的平均值进行补偿。这一措施是为了确保数据的连续性和准确性。

•为了保证数据的完整性，记录所有出现车辆缺失的帧数，并确保这些缺失帧数不超过总帧数的 10%。一旦超过此比例，可能需要重新处理或调整检测算法。

•结果保存与人工核查：为了确保最终数据的准确性，我们将处理过的图像帧保存到本地。其中，最多保存 3 张包含检测框的图像，这些图像将用于最后的人工抽检，以核定 YOLO 车辆检测的结果。

道路长度的计算：在完成视频数据处理并获得每帧视频中被识别车辆的平均数量后，进一步估算摄像头所覆盖的范围距离。

具体步骤如下：

•计算平均车辆数量，通过视频处理获得的每帧平均车辆数量为 N_{avg}

•估算摄像头识别范围，利用已知的每辆车的平均长度 L_{car} 和车辆之间的安全间隔 d_{safe} ，可以估算出摄像头所覆盖的道路长度 L_{road}

交通密度定义为单位道路长度内的车辆数量，通常以“辆/公里”为单位。在估算了道路长度 L_{road} ，通过公式 4-1 计算每米的车辆数量：

$$K = \frac{N}{L_{road}} \tag{4-1}$$

平均车速的计算：分别使用 YOLO 模型检测车辆的位置并计算位移和结合多目标跟踪算法来更精准地计算速度。而对于利用 YOLO 计算车速，位移距离由于缺乏车辆在世界坐标系中的精确位置以及像素与现实世界之间的尺度转换关系，我们无法通过传统的射影几何方法来计算车辆的实际移动距离。这里采取另一种方法：即将车辆从首次被检测到直至最后一次被检测到的路径长度，视为车辆的移动距离，基于此位移与时间间隔来估算车辆的平均速度。具体公式如 4-2 所示：

$$V = \frac{L}{FPS \times \mathrm{T_f}} \tag{4-2}$$

相比单纯依赖 YOLO 的检测结果，使用多目标跟踪（SORT）算法能够更稳定地跟踪车辆的位置，从而更精准地计算车速。SORT 算法是一种基于卡尔曼滤波的跟踪算法，它可以在每帧之间保持车辆的轨迹，极大减少丢失车辆的情况。通过 YOLO 获取初始检测结果，将其输入到 SORT 跟踪器。SORT 通过卡尔曼滤波器预测车辆的运动轨迹，并结合检

测结果更新车辆的位置。

交通流量的计算：我们采用经典的交通流理论公式 4-3 计算

$$Q = V \cdot K \tag{4-3}$$

4.3 第一小问

针对题目提供的视频数据，通过对四个观测点的车流进行统计分析，从数据预处理已经提取了每个观测点在不同时间段的交通流参数，包括车流量、平均车速，观测道路长度等。为了捕捉流量随时间的精细变化，采用 10min 为一个时间窗口的粒度进行统计分析，从而获得流量的短期波动情况。

4.3.1 第一个监测点

对于第一个监测点，我们利用两种不同的算法对速度进行计算，从而获得不同的速度值与流量值。具体的两个算法在视频速度和流量关于时间的对比如图 4-2 所示：

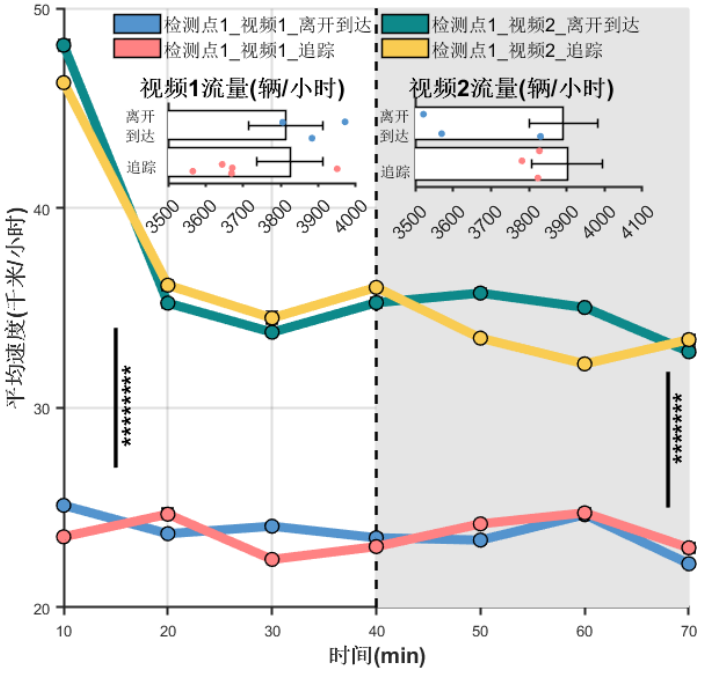

图 4-2 追踪算法与单 yolo 算法的速度与流量对比

图 4-2 的速度曲线中追踪算法（黄色曲线）相比于单 YOLO 算法能够捕捉到更平滑且持续的速度变化，标准差更小，鲁棒性更好。单 YOLO 算法在某些时段出现速度的急剧下降，可能由于单帧检测时无法持续追踪到同一辆车，导致速度估计不稳定。在流量的计算中，追踪算法（黄色曲线）保持了较为平稳的车辆流量估计，而单 YOLO 算法由于只依赖于逐帧检测，流量估计呈现波动更大。追踪算法可以在多帧之间连续跟踪车辆，从而获得更准确的流量估计。所以最后使用追踪算法来预测其他视频的平均速度与交通流量。[2]

从这幅图中可以观测到当前的交通流量是分布在 3500-4100 辆/小时之间，速度范围约在 20km/h-50km/h 之间，而对于高速公路的通行能力通常为 4400 辆/小时，平均速度在 100km/h。推断出当前节假日高速公路处于拥堵状态，应急车道的启用对于解决拥堵起决定性作用。

对于检测点 1 的不同视频的车辆密度如图 4-3 所示：

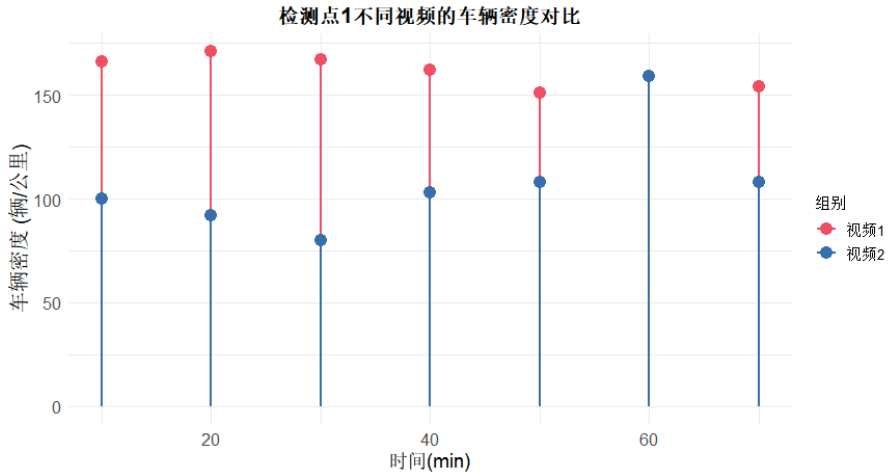

图 4-3　检测点 1 不同视频的车辆密度对比

可以观察到，视频 1 的车辆密度明显高于视频 2，这表明视频 1 捕捉到的时间段（午高峰期间）的车流量更大，符合高峰期的交通特征。尤其是在午高峰时段，车辆的密集程度大幅上升，导致单位距离内的车辆数目增加。相比之下，视频 2 覆盖的时间段或道路状况则较为通畅，车辆分布更加稀疏，密度相对较低。通过该检测点 1 的分析，对于应急道路的启动时间段有了更清楚的判断与决策。

4.3.2　第二个监测点

对于检测点 2，首先绘制了在节假日期间不同视频下平均速度随时间变化的图像，如图 4-4 所示。

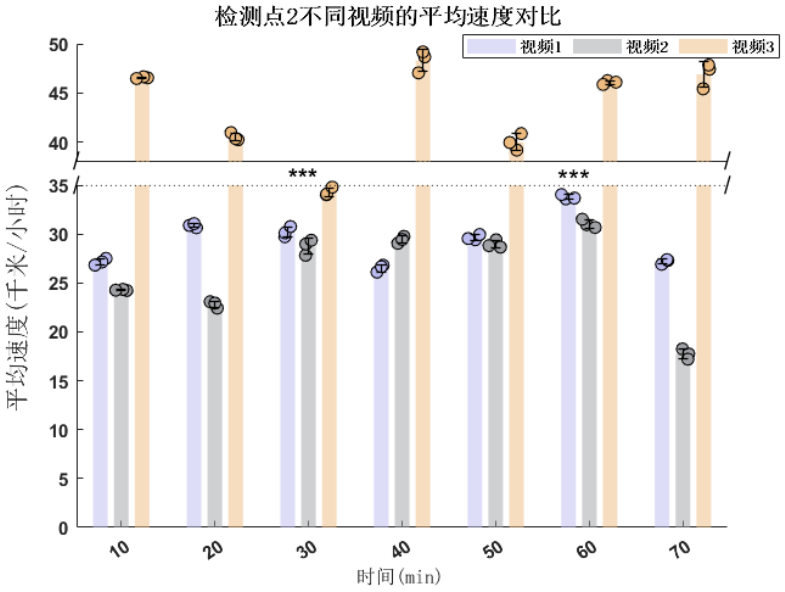

图 4-4　检测点 2 不同视频的平均速度对比

图 4-4 中，不同视频在各个时间段内的速度表现有明显差异。可以观察到视频 3 的速度通常高于视频 1 和视频 2。结合视频中的现实时间可以观察到，视频 1 从上午 11 点到 13 点，视频 2 从 13 点到 14 点，视频 3 从 14 点到 15 点左右，视频 1 处于交通高峰时段，许多车辆可能因为午餐或出行等原因增加，导致道路上的车辆流量增大，因此车速有所下降，因此在应急车道的启用时段优先考虑交通高峰时间段。

检测点 2 不同视频下的交通流量与车辆密度的关系如图 4-5 所示：

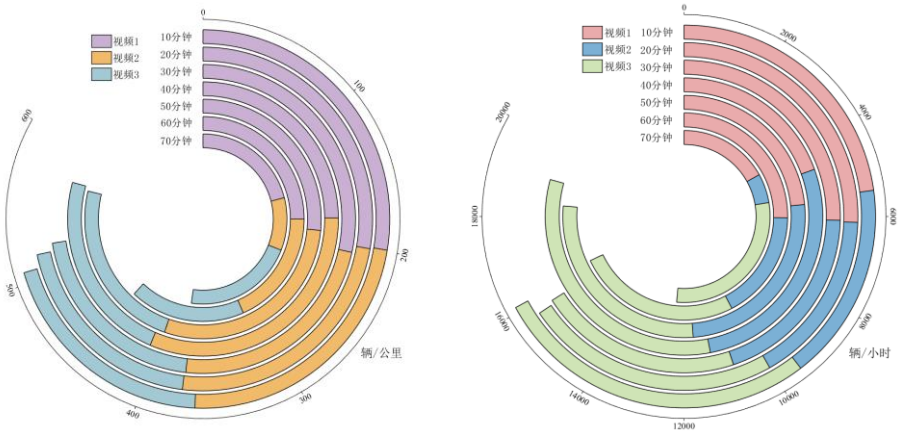

图 4-5　检测点 2 不同视频的密度与流量

左图展示了检测点 2 的交通密度（辆/公里），右图展示了流量（辆/小时）。不同颜色代表不同视频，环形代表不同时间段的变化。

交通密度（左图）：从图中发现视频 1 的交通密度显著高于其他视频，特别是在最初的时间段内，表明该时段内车辆较为集中，车距较小。随着时间的推移，交通密度逐渐下降，这与典型的午高峰交通模式相符。在视频 2 和视频 3 中，交通密度呈现持续下降的趋势，表明随着时间的推移，车辆的间距逐渐加大，车辆运行速度也在提高。这与之前提到速度的上升趋势相吻合，说明在非高峰时段，交通更加顺畅。

交通流量（右图）：可以观察到视频 1 的在最初 10s 的交通流量达到了 5800 辆/小时，而高速公路的双车道最大服务交通量 4400 辆/小时，高交通流量必然会导致交通拥堵，尤其是在双车道无法容纳如此多车辆的情况下。因此，在节假日期间，启用应急车道对于缓解交通压力、提高道路通行能力至关重要。

4.3.3　第三个和第四个监测点

对于第三个和第四个检测点的交通流量、平均速度、车辆密度的数据如图 4-6 所示。

图片第一行是两个检查点不同视频的流量随时间变换的图像。每隔 10min 采集一次数据，可以发现检测点 3 的第 60min 左右，在 12：30 时间段，交通量最大，同理检测点 4 的第 60min 左右，也出现高峰交通量，且大部分时间段的交通量都大于最大服务交通量，说明会出现交通拥堵的状况，需要采取相应的措施疏导交通。

图片第二行是两个检查点不同视频的平均速度随时间变换的图像。该平均速度是采样点 10min 的平均速度，而非当前完整视频的平均速度。

图片第三行是两个检查点不同视频的车辆密度随时间变换的图像。交通密度反映了单位距离内的车辆数量，从图中可以观察到在流量高的时间段，交通密度也随之增加，尤其是在视频 1 中，密度峰值出现在流量高峰期。随着时间的推移，密度逐渐下降，这表明车辆间的距离增大，交通状况得以缓解。

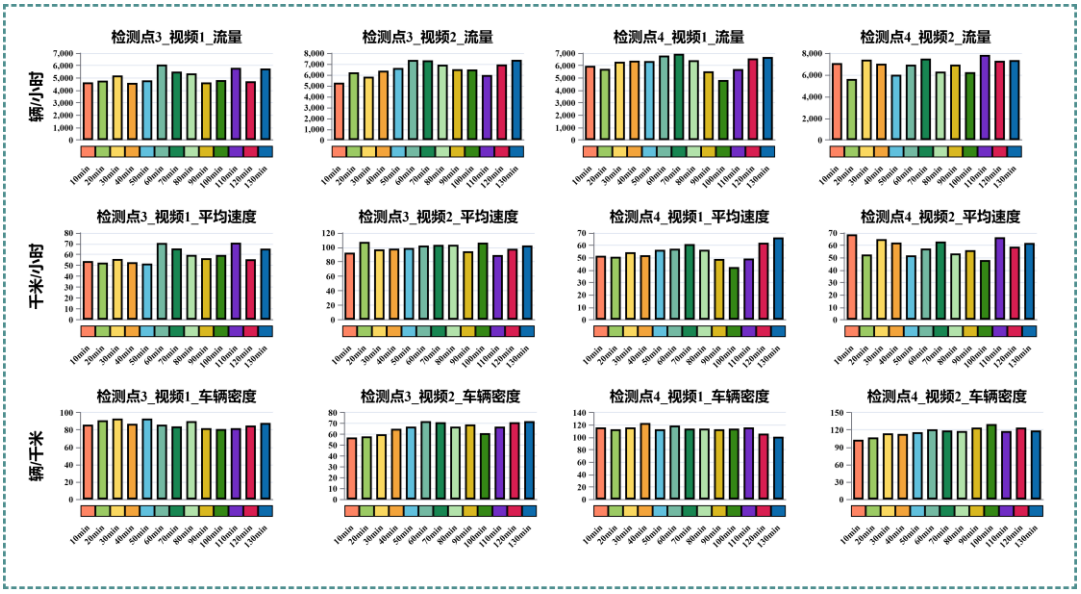

图 4-6　三、四检测点的流量、速度和密度

4.4　第二小问与第三小问

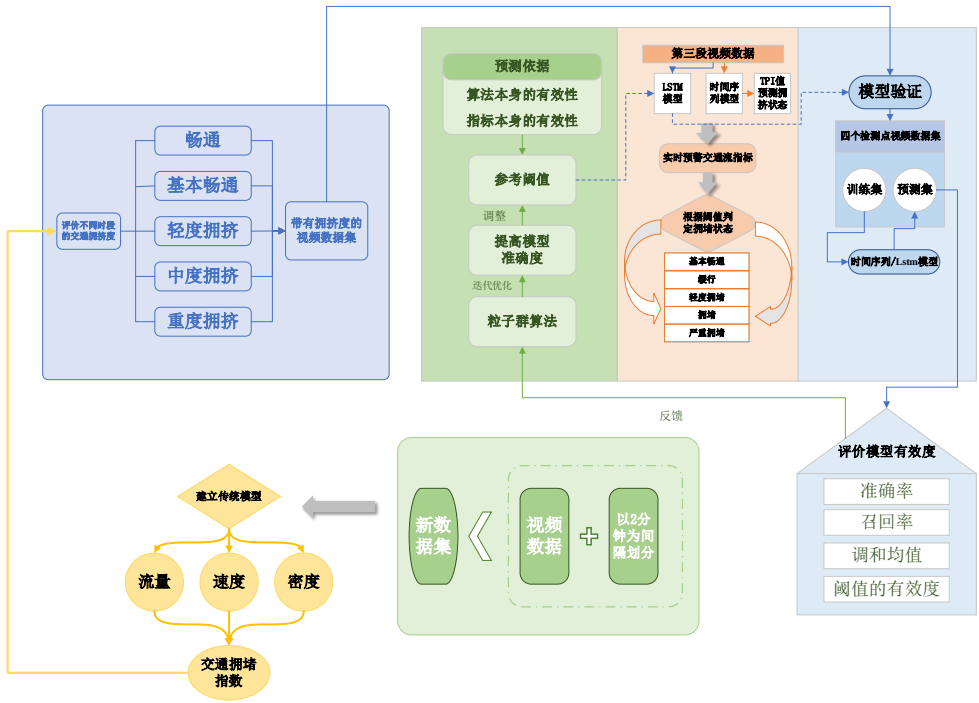

图 4-7　第二小问与第三小问的技术路线图

4.4.1　数据再处理：

为了构建更加精确的交通流拥堵模型，本团队以两分钟为时间间隔，对视频数据集进行了重新划分，以获取更加细致且准确的交通参数。通过这一方法，从四个观测点收集了车流密度、流量、速度以及道路长度等关键信息，进而组成了一个全新的数据集。

4.4.2　传统交通拥堵模型的建立：

本团队根据郑淑鉴，杨敬锋的[3]研究分别建立了两种交通拥堵指标模型：一种是基于

交通量计算的交通拥堵指标模型，另一种是基于出行时间比计算的交通拥堵指标模型。

对于基于交通量计算的交通拥堵指标（Degree of Congestion， DC）这一方法。该方法源自日本的交通研究，旨在量化特定路段的拥堵情况。定义为某路段实际交通量与评估基准量之比，通过公式 4-4 计算：

$$DC = \frac{\omega \times Q}{C} \tag{4-4}$$

其中：

- ω 为权重系数，用以区分不同类型车辆的影响：小型车辆的权重为 1（$\alpha = 1$）大型车辆的权重为 α（α 根据具体标准车当量系数确定）。
- Q 为实际的交通量。
- C 为基准交通量，通常根据道路的规划容量、设计通行能力、峰值车流量、同方向车道数量等多项因素求出。

表 4-2　拥堵度代表的道路运行水平

拥挤度 *DC*	道路的拥挤程度
$DC < 1$	畅通
$1 \le DC < 1.75$	拥堵时段逐渐增加
$DC \ge 1.75$	慢性拥堵

通过对 4 个检测点的数据进行公式 4-4 计算，得到 *DC* 的值，绘制基于平均速度和交通流量两个关键参数的拥挤程度。

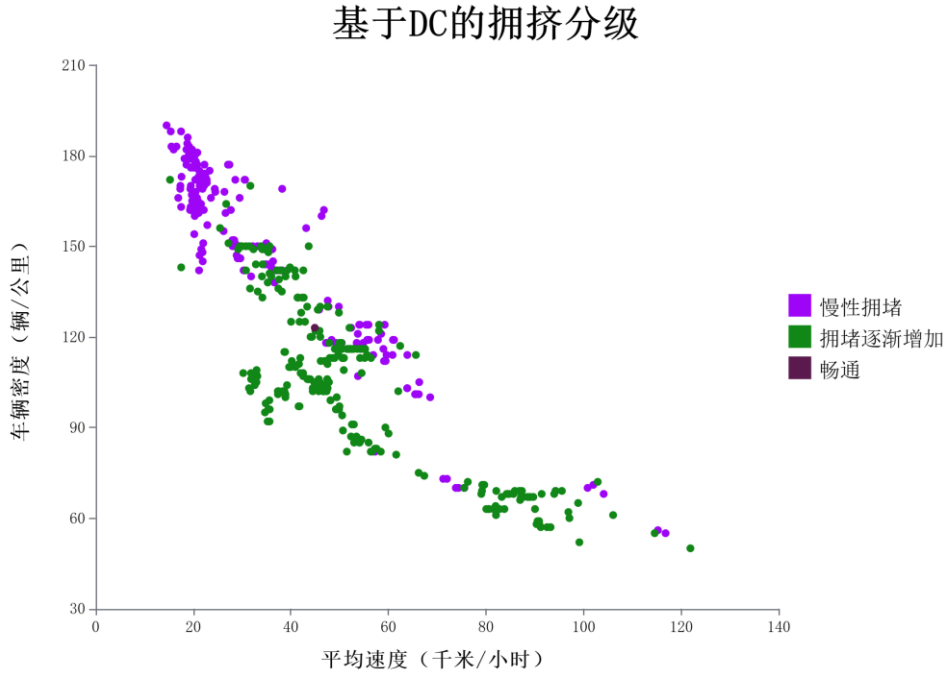

图 4-8　基于交通量的速度-流量-拥堵情况分级

慢性拥堵出现在平均速度较低（约 20-40 千米/小时）且车辆密度较大的区域。在平均

速度 40-80 千米/小时，车辆密度出现在 80-150 辆/公里区间，拥堵逐渐增加。该图像反映了速度、车辆密度与拥堵程度有强相关性。

在构建基于出行时间比的交通拥堵指标模型时，本团队参考了深圳市 2012 年发布的城市交通运行指数。这一指标通过出行时间比的计算方法来衡量道路或路网的交通拥堵情况。

出行时间比（*TPI*）的计算公式为[4]：

$$TPI = F(R_T) \tag{4-5}$$

R_T 为特定路段内的行程时间比； $F()$ 为专家打分确定的转换关系。

行程时间比 R_T 由路网中一次出行平均花费的实际时间和期望速度下的行程时间计算得出。其具体计算公式为：

$$R_T = \frac{\overline{T}}{\overline{T}_d} = \frac{\sum_{k=1}^{n}(q_k \times t_k)}{\sum_{k=1}^{n}(q_k \times t_{dk})} = \frac{\sum_{k=1}^{n}(q_k \times l_k / v_k)}{\sum_{k=1}^{n}(q_k \times l_k / v_{dk})} \tag{4-6}$$

- q_k 为道路 k 的流量。
- t_k 为实际行程时间。
- l_k 为道路 k 的长度。
- v_{dk} 为期望速度。

表 4-3　道路交通运行指数分级

指数范围	拥堵等级
0～1	畅通
1～2	基本畅通
2～3	缓行
3～4	轻度拥堵
4～5	拥堵
＞5	严重拥堵

传统拥挤模型的分类结果如图 4-9 所示：

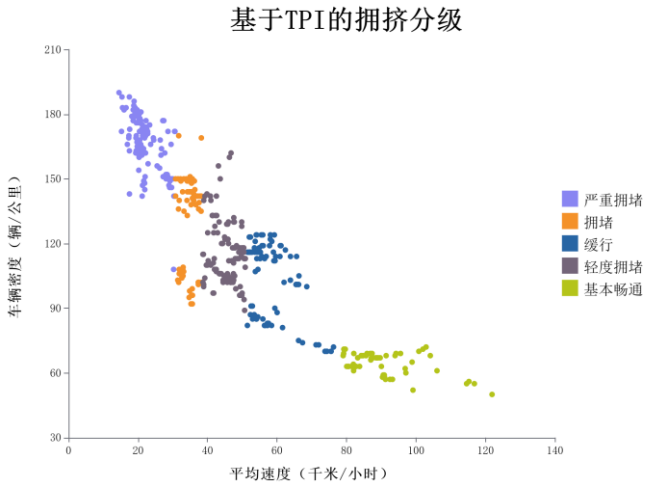

图 4-9　基于出行时间比的速度-流量-拥堵情况分级

从图中可以看出，交通拥堵情况的分级与平均速度和交通流量有关，与车辆速度的相关系数更大。严重拥堵的情况集中在平均速度较低（20-40 千米/小时）的区间；拥堵状态分布在平均速度 40-60 千米/小时区间，流量较高；在平均速度 60-80 千米/小时，道路处于缓行状态；轻度拥堵发生在平均速度 80-100 千米/小时之间；平均速度超过 100 千米/小时时，道路基本处于畅通状态，流量较低，车流比较畅通。

考虑到应急车道的开启需要提前 5~10min 做好准备，结合上述两个传统的交通拥堵评价模型，选取如表 4-4 所示的起始阈值，为建立模型提供参考阈值。

表 4-4　实时预警模型参考阈值

拥堵等级	平均速度（千米/小时）	交通流量（辆/小时）
基本畅通	90.92	5812
缓行	58.60	6020
轻度拥堵	45.50	5175
拥堵	34.83	4575
严重拥堵	22.26	3680

4.4.3　实时交通预警模型的构建

本团队选用时间序列模型和 LSTM 模型，并通过粒子群算法对参考阈值进行动态更新和调整，以获得更优的阈值参数。这两种模型相互验证，共同构建了一个实时交通预警系统。

预测理论依据：

时间序列模型的有效性体现在其对高频数据的处理能力上。团队对视频数据进行每 2 分钟一次的采样，这种高频数据能够更好地捕捉交通流量的动态变化，尤其是在交通拥堵突发的情况下，时间序列模型能够提供更为及时和准确的预测。它专注于历史数据的趋势和周期性，在短期预测未来交通流量时表现出极高的准确性。[5]

LSTM 模型的有效性，对于每 2 分钟采集一次的数据，LSTM 可以很好地捕捉到长期趋势以及短期波动，尤其在面对非线性变化时，能够表现出卓越的预测能力。交通流量数据通常具有非线性特征，并且可能会受到突发事件的影响。LSTM 模型通过其门控机制，

能够在这种情况下仍然保持较高的预测精度，识别出数据中的复杂模式，进而准确预测未来的交通状况。另外，LSTM 模型可以通过不断学习新的数据（每 2 分钟采集的新数据），逐步提升其预测性能。这种适应性使得 LSTM 模型能够随着交通流量模式的变化而进行自我调整，从而提供更具动态性的预测结果。

与其他常见线性回归模型，决策树/随机森林模型相比的有效性如表 4-5 所示

表 4-5　时间序列算法、LSTM 与常用算法的对比

方法	方法描述	优缺点
时间序列算法	分析时间序列数据的历史趋势和周期性来预测未来的值。于预测时间相关的指标	适用于稳定的时间序列数据
LSTM	长短期记忆网络，擅长处理和预测序列数据，尤其是在长期依赖关系的情况下。	对复杂的时序数据有很强的建模能力，能够捕捉长期依赖关系
决策树算法	根据数据的属性采用树状结构建立决策模型。	可能存在过拟合倾向，且对噪音敏感。
集成算法	由多个学习模型独立训练后，通过加权方式集成预测结果以提高整体模型性能。	复杂性高，训练时间长，解释性较差。

数据的有效性：对第三个检测点每 2min 的采样点的数据进行相关性分析，来证明数据与交通拥堵是具有相关性。因为是离散点，我们采用了皮尔逊相关性分析，衡量两个变量之间的线性相关性程度，结果以相关系数（r）的形式呈现。

计算皮尔逊相关系数：

$$r=\frac{\sum(X_i-\bar{X})(Y_i-\bar{Y})}{\sqrt{\sum(X_i-\bar{X})^2\sum(Y_i-\bar{Y})^2}} \tag{4-7}$$

X_i为交通流量（或速度、密度）数据点；Y_i为拥堵程度数据点。$\bar{X}$ $\bar{Y}$ 分别为对应数据的均值。

各个数据之间与交通拥堵程度的结果如表 4-6 所示

表 4-6　交通流参数相关性

	拥堵等级	**显著性**
平均速度(千米/小时)	-0.8378	$5.2327e^{-19}$
交通流量(俩/小时)	-0.2072	$9.0040e^{-02}$
车辆密度(俩/小时)	0.8571	$1.1040e^{-20}$

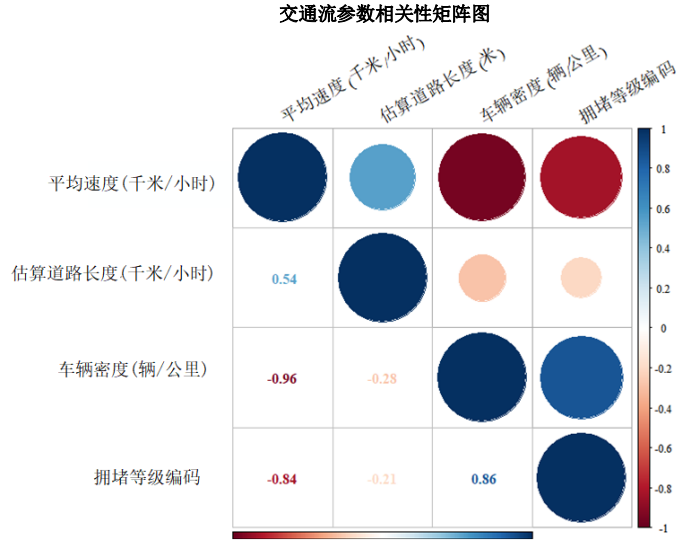

图 4-10　交通流参数相关性矩阵图

从表和图中分析平均速度与拥挤等级的关系：由于 p 值非常小（远小于 0.05），可以认为这个负相关性非常显著。当车辆的平均速度降低时，拥挤等级上升，与现实情况相符。

交通流量与拥挤等级的关系：交通流量与拥挤等级之间的负相关性较弱，这表明交通流量并不是单一决定拥挤等级的因素。交通流量与拥挤的关系可能受到其他因素的干扰，如道路容量、车速等。p 值略大于 0.05，表明这个相关性不够显著，因此在预测拥挤等级时，仅考虑交通流量的效果可能有限。

车辆密度与拥挤等级的关系：车辆密度与拥挤等级之间存在非常强的正相关性。随着车辆密度的增加，拥挤等级也显著上升。车辆密度高通常意味着道路空间紧张，易引发拥堵。p 值极小，说明正相关性非常显著。车辆密度是影响拥挤等级的一个关键因素。三者的相关性分析，与前文的传统交通拥挤模型的图示相同，两者相互印证。

参考阈值自更新与调节[6]：

依据前文的传统交通拥挤模型得到初始的阈值，在实际应用中若长期采用固定阈值，随着道路环境的变化，算法的检测性能将会越来越低，因此必须对阈值进行实时自适应调整。在粒子群算法优化阈值时，参考程小洋[7]等研究设置阈值更新条件：

- 流量变化大时，无论是否检测到事件，都需要进行阈值更新。
- 流量变化小时，若检测到事件，则需要进行阈值更新。
- 路段属性变为匝道、隧道或弯道时，应进行阈值更新。
- 天气情况发生变化时，需要进行阈值更新。

为了实现交通事件检测阈值的实时自适应调整与优化，必须选择一种运算速度快且具有全局寻优能力的求解算法。因此，本研究选用粒子群优化算法来优化交通事件检测阈值模型。

粒子群优化算法（ParticleSwarmOptimization，PSO）是一种基于群体协作的随机搜索算法，最早由 Eberhart 博士和 Kennedy 博士提出。该算法通过模拟鸟群觅食行为来寻求问题的最优解，不依赖于被求解问题的数学模型，具有很强的自适应性。PSO 算法将问题的最优解作为搜索方向，通过群体中的个体（即“粒子”）协作，不断迭代寻求全局最优解。

在 PSO 中，每个粒子的位置表示一个潜在解，目标函数的适应度值衡量该解的优劣。

粒子群体通过更新速度和位置，逐步逼近最优解。每个粒子的历史最佳位置和全局最佳位置共同决定其下一步的搜索方向。

粒子的速度和位置更新公式 4-8：

$$v_{id}^{k} = \omega v_{id}^{k-1} + c_1 r_1 (pbest_{id} - x_{id}^{k-1}) + c_2 r_2 (gbest_d - x_{id}^{k-1}) \tag{4-8}$$

$$x_{id}^{k} = x_{id}^{k-1} + v_{id}^{k}$$

- ω 是惯性因子，控制搜索空间的范围；
- $c_1 c_2$ 是加速常数，调节学习的步长；
- $r_1 r_2$ 是取值范围为[0，1]的随机数，增加搜索的随机性。

实时阈值的求解流程：

为了实现阈值的实时更新与优化，我们在初始阈值确定方法的基础上，提出了阈值寻优时间窗的概念，以动态调整实时阈值。具体步骤如图 4-11 所示：

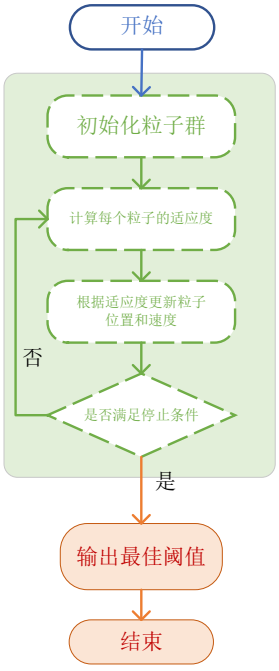

图 4-11　粒子群算法优化阈值流程图

4.4.4　基于时间序列算法建立拥堵预警模型

在交通流量预测和拥堵预警中，时间序列算法由于其对时间依赖性数据的良好处理能力，成为构建拥堵预警模型的重要工具。通过对历史交通数据的分析，时间序列算法能够识别出交通流量的趋势和周期性变化，从而实现对未来交通状态的有效预测，并提前发出拥堵预警。

本文采用 ARIMA 模型来构建拥堵预警模型，ARIMA 模型是一种线性时间序列模型，结合了自回归（AR）、差分（I）和移动平均（MA）三种模型的特点。ARIMA 模型通过对时间序列数据的自相关性和移动平均特性进行建模，可以在一定程度上捕捉数据中的趋势和周期性变化，从而实现对未来值的预测。ARIMA 模型的基本形式为：

$$Y_t = \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \cdots + \phi_p Y_{t-p} + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \cdots + \theta_q \epsilon_{t-q} + \epsilon_t \tag{4-9}$$

- Y_t 是时间序列在时间 t 的值；
- $\phi_1,\phi_2,\cdots,\phi_p$ 是 AR 部分的系数；
- $\theta_1,\theta_2,\cdots,\theta_q$ 是 MA 部分的系数；
- ϵ_t 是时间 t 的白噪声误差；
- p 和 q 分别是 AR 和 MA 部分的阶数；
- d 是差分次数，用于处理非平稳时间序列，使其转化为平稳序列。

为了构建拥堵预警模型，本文采用以下步骤进行 ARIMA 模型的构建与应用：

数据差分与平稳化：对非平稳时间序列进行差分处理，使其转化为平稳序列，便于模型的建模和预测。

模型识别：通过分析自相关函数（ACF）和偏自相关函数（PACF）图，确定 AR 和 MA 部分的阶数p和q，以及差分次数d。

参数估计：使用最大似然估计法或最小二乘法估计模型参数ϕ和θ，使模型与历史数据的拟合度达到最优。

模型检验：通过残差分析（例如白噪声检验、Ljung-Box 检验等）检验模型的适用性，确保残差序列为白噪声，即模型能够较好地捕捉数据的动态特性。

模型预测：在模型构建完成后，利用 ARIMA 模型对未来的交通流量、车辆密度、车辆速度进行预测。

在建立基于时间序列算法建立拥堵预警模型，首先绘制了不同交通流参数（车辆密度、交通流量和平均速度）ARIMA 模型在不同阶数组合下的 AIC（AkaikeInformationCriterion）值。AIC 值代表统计模型优劣的重要指标，值越小表示模型的拟合效果越好。

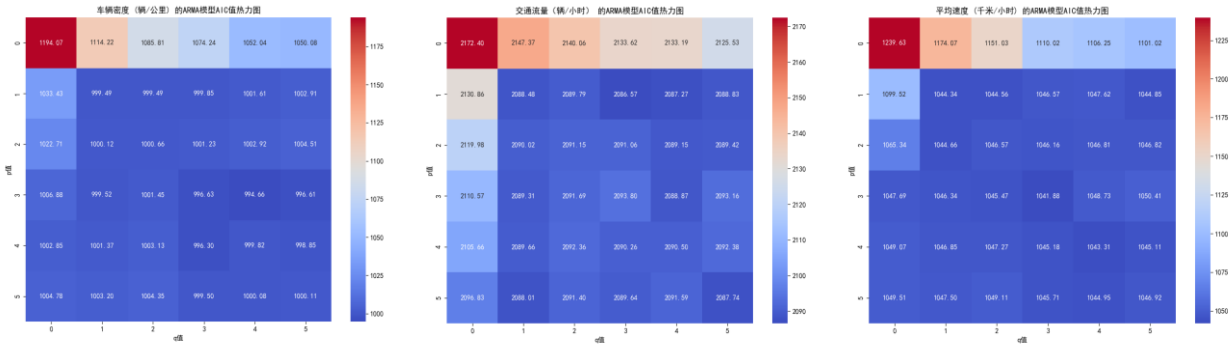

图 4-12　不同交通流参数 AIC 值

车辆密度（辆/公里）的 ARIMA 模型 AIC 值热力图：在不同阶数组合下，AIC 值存在明显的差异。车辆密度的 AIC 值在(p =4， d =1， q =2)和(p =4， d =1， q =3)组合下相对较低，说明这些组合的模型在拟合车辆密度数据时表现较好。

交通流量（辆/小时）的 ARIMA 模型 AIC 值热力图：交通流量的 AIC 值在(p =4， d =1， q =1)和(p =4， d =1， q =2)组合下较低，显示这些组合的模型对交通流量数据的拟合效果较优。

平均速度（千米/小时）的 ARIMA 模型 AIC 值热力图：对于平均速度数据，AIC 值在(p =4， d =1， q =1)和(p =4， d =1， q =2)组合下表现出最低值，这表明这些组合在拟合平

均速度数据时具有最佳性能。

通过对不同交通流参数的 ARIMA 模型 AIC 值的分析，可以看出在(p =4，d =1，q =1)和(p =4， d =1， q =2)的组合下，无论是车辆密度、交通流量还是平均速度，这些模型均表现出较好的拟合效果。因此，基于这些参数组合的 ARIMA 模型在实际应用中具有很高的预测价值，可以帮助准确预警交通拥堵情况。

时间序列预测结果图中，观察到 ARIMA 模型预测的未来的交车辆密度、交通流量和平均速度。

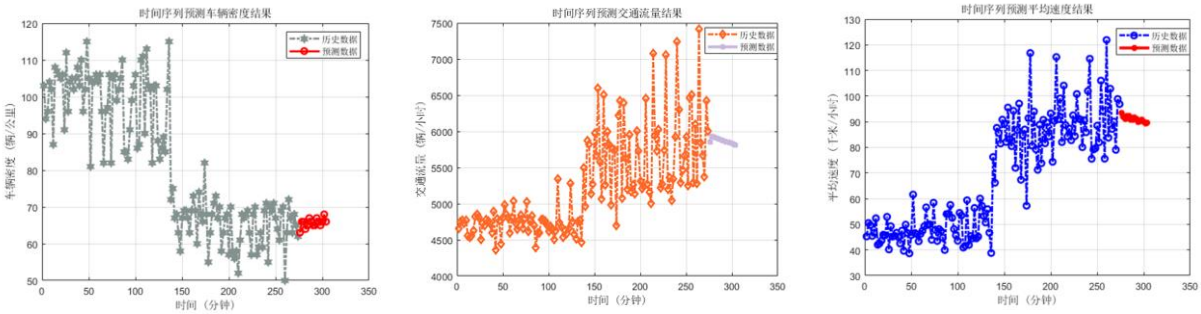

图 4-13　不同交通流参数预测结果

时间序列预测车辆密度结果：

历史数据：图中灰色部分表示过去车辆密度的历史数据，车辆密度在不同时间段波动较大。

预测数据：红色部分为 ARIMA 模型对未来车辆密度的预测结果。可以观察到预测数据相对稳定，集中在较窄的区间内，与历史数据相比略有平滑，保持在 60-70 辆/公里。

时间序列预测交通流量结果

历史数据：橙色部分显示了过去的交通流量数据，存在明显的高峰和低谷。

预测数据：紫色部分表示未来交通流量的预测。预测数据相对稳定，流量在 5500-6000 辆/小时。

时间序列预测平均速度结果

历史数据：蓝色部分为过去的平均速度数据，显示出显著的波动，尤其是在特定时间段内。

预测数据：红色部分表示未来的平均速度预测。在 300min 左右，速度降为 90km/h 之下。

时间序列分析结果：在第三个检测点的视频结束的 30min 后，出现了轻度拥堵现象。

4.4.5　基于时间序列分析的 TPI 的交通拥堵模型预测：

在交通流量管理中，时间序列模型如 ARIMA 已经被广泛用于交通参数（如流量、速度、密度等）的预测。然而，随着交通状况的复杂性增加，仅预测交通参数往往难以全面反映交通拥堵的实际情况。因此，结合时间序列模型进行 TPI（出行时间比）的预测，提供了一个更为综合和改进的预测方法。以下是基于 TPI 预测与传统交通参数预测的对比分析，以突出这种改进的效果。

传统的交通预测模型通常聚焦于流量、速度和车辆密度等单一参数。这些参数能够反

映某一方面的交通状态，但难以捕捉复杂的交通拥堵情境。例如：

流量预测：流量预测有助于识别高峰期和低谷期，但不能直接反映出行时间的延误。

速度预测：速度预测能够显示车速的波动情况，但对预测整体交通系统的效率贡献有限。

车辆密度预测：车辆密度预测可以揭示拥堵的潜在风险，但不足以单独判断出具体的拥堵程度。

TPI 预测：TPI 结合了多种交通参数，计算了实际出行时间与期望出行时间的比值，因此能够更直接地反映出交通流的运行效率。TPI 值不仅考虑了交通流量和速度，还反映了车辆在路网上的整体通行时间状况。因此，相对于单一参数预测，TPI 预测能够更准确地预示出实际拥堵状况。

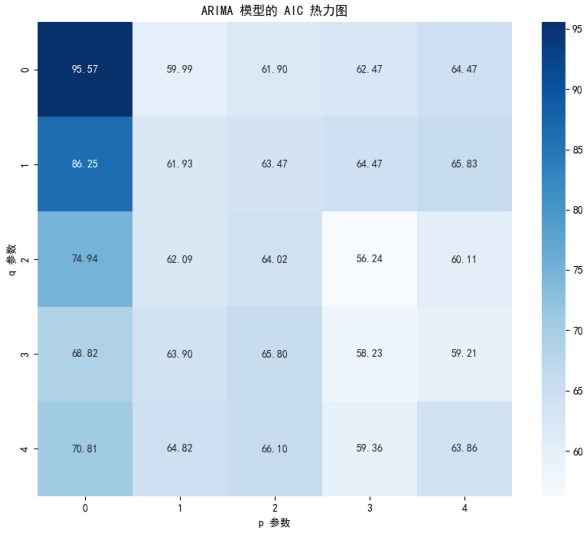

图 4-14　TPI-AIC 热力图

ARIMA 模型*TPI*的 AIC 热力图，从热力图中可以看出，参数组合(p =1， q =3)处的 AIC 值最低（56.24），表明在该组合下，ARIMA 模型对数据的拟合效果最佳。因此，(p =1，q =3)作为 ARIMA 模型的参数是一个合理的选择。随后对第三个视频进行未来 30 分钟的*TPI*预测分析，结果如图 4-15。

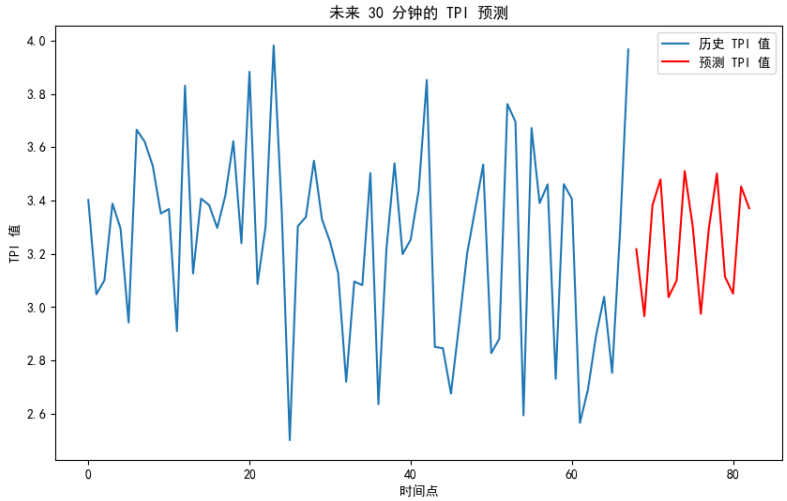

图 4-15　未来 30 分钟的 TPI 的预测

TPI 的历史值与未来 30 分钟的预测值。蓝色曲线表示历史 TPI 值，红色曲线表示 ARIMA 模型对未来 30 分钟的预测结果。模型很好地预测了 TPI 的总体趋势，预测值较为平稳，从图中可以观察到 TPI 在 3.0 与 3.6 之间波动，根据表 4-2 可以预测到未来 30min 的交通拥挤程度在轻度拥堵与缓行之间波动。

通过使用 *TPI* 值进行时间序列预测，模型不仅能够识别流量和速度的趋势，还能够综合这些因素来评估交通系统的整体运行效率。TPI 预测能更直接地与拥堵状况挂钩，因此能够更及时、更准确地反映出潜在的交通问题。ARIMA 模型用于预测未来 30 分钟的 *TPI* 值，成功预示了“轻度拥堵”和“缓行”等实际交通状况，这比单纯的流量或速度预测更加直观且有针对性。

相对于传统的交通参数预测，基于 *TPI* 的时间序列预测模型具有更高的准确性和更强的短期波动应对能力。这一改进方法能够综合多种交通因素，为交通管理提供更全面、更直接的决策支持。*TPI* 预测不仅可以准确预示交通拥堵状况，还能够提升交通管理系统的响应速度和有效性。

4.4.6　基于 LSTM 模型建立拥堵预警模型

在交通流量预测和拥堵预警领域，传统的时间序列模型（如 ARIMA）在捕捉长期趋势和周期性变化方面表现良好，但在处理复杂的非线性动态变化和长时间依赖性时可能存在一定的局限性。为了解决这一问题，本研究采用了长短期记忆网络（LSTM）模型，旨在构建一个更为精确的交通拥堵预警系统。

LSTM 模型概述：

LSTM（Long Short-Term Memory）是一种特殊的递归神经网络（RNN），能够有效处理序列数据中的长短期依赖问题。与传统的 RNN 不同，LSTM 通过引入门控机制（输入门、遗忘门和输出门）来控制信息的流动，使其能够在长时间跨度上保持对关键信息的记忆。因此，LSTM 模型特别适合用于预测复杂的时间序列数据，如交通流量、车辆密度和出行时间比（TPI）等。

LSTM 模型的构建与训练：

输入层：LSTM 模型的输入层接收多维度的交通数据，历史交通流量、速度、密度。

LSTM 单元结构公式：

遗忘门：决定先前状态信息的保留程度

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{4-10}$$

输入门：决定当前输入信息的重要性并更新记忆单元状态：

$$\begin{aligned} i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\ \tilde{C}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \end{aligned} \tag{4-11}$$

记忆单元状态更新：结合遗忘门和输入门的结果更新记忆单元状态：

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \tag{4-12}$$

输出门：决定哪些信息会影响当前的隐藏状态，并通过当前的记忆单元状态生成隐藏状态：

$$\begin{aligned} h_t &= o_t \cdot \tanh(C_t) \\ o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \end{aligned} \tag{4-13}$$

损失函数：使用均方误差（MSE）作为损失函数，衡量模型预测值与真实值之间的差距。

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{4-14}$$

其中，$\hat{y}_i$ 是模型的预测值，y^i 是真实值，n 是样本数量。

优化器：采用 Adam 优化器进行梯度下降优化，调整模型参数以最小化损失函数。

训练过程：将数据集分为训练集和验证集，使用训练集对模型进行训练，并通过验证集进行调优，以防止模型过拟合。

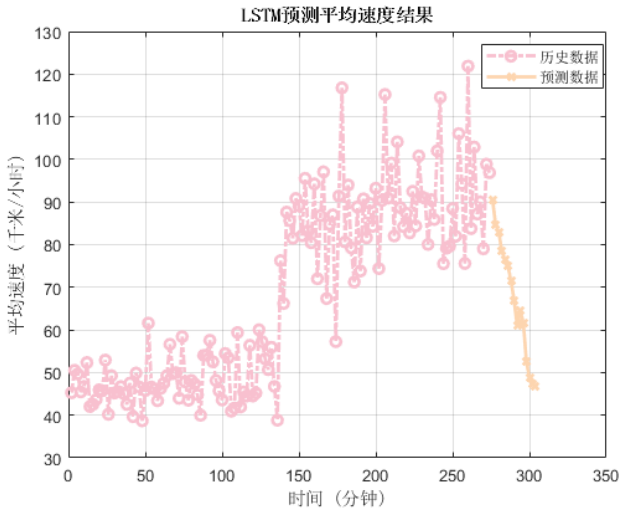

图 4-16　LSTM 预测平均速度结果

LSTM 预测平均速度结果：图中展示了 LSTM 模型对未来平均速度的预测结果（橙色），速度在发生急剧下降，说明未来可能产生交通拥堵。LSTM 模型在长时间跨度内的速度变化趋势时表现良好，尤其是在预测未来速度下降的过程中，模型给出了相对准确的预警信息。这表明 LSTM 能够有效处理交通流中的非线性动态变化。

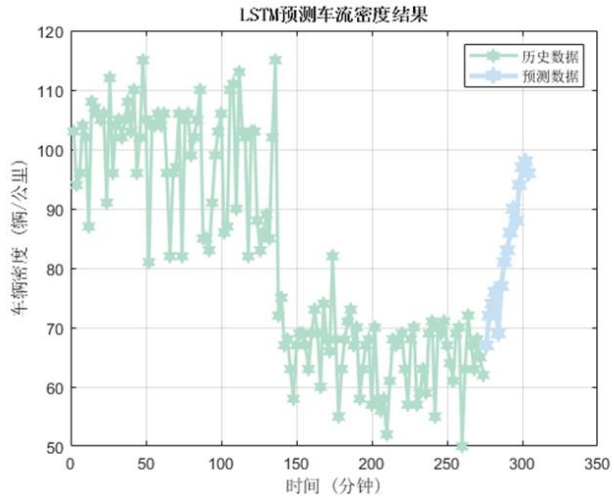

图 4-17　LSTM 预测平车流密度结果

LSTM 预测车流密度结果：该图展示了 LSTM 模型对未来车流密度的预测（蓝色）与历史数据（绿色）的对比，以及 LSTM 预测在未来 50min 的车流密度激增，说明未来会发生交通拥堵。

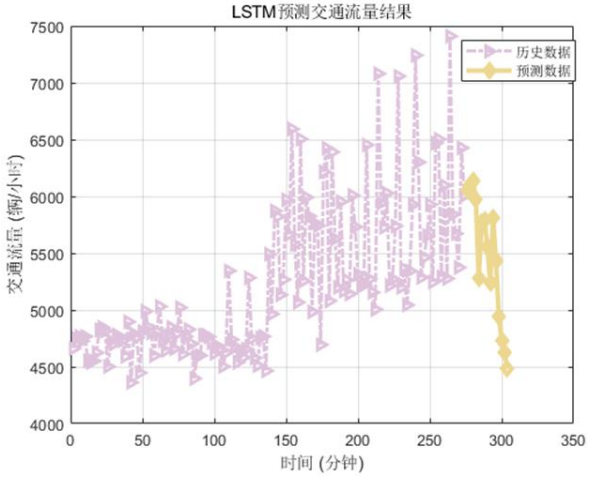

图 4-18　LSTM 预测交通流量结果

LSTM 预测交通流量结果：此图显示了 LSTM 模型对未来交通流量的预测结果（黄色）与历史数据（紫色）的对比预测未来流量下降的趋势，交通管理系统可以提前采取措施来缓解潜在的拥堵。

4.4.7　自适应阈值调节与报警函数设计

报警函数 Alert(t)是我们交通拥堵预警系统的核心。通过判断交通流量和车流密度是否超过设定的阈值和，函数输出 1 表示触发预警，输出 0 表示不触发预警。

$$Alert(t)=\begin{cases}1, & \text{if } v(t)>v_{\max} \text{ and } k(t)>k_{\text{crit}} \\ 0, & \text{otherwise}\end{cases} \tag{4-15}$$

然而，固定的 $v_{\max}$ 和 k_{crit} 阈值无法适应复杂的、不断变化的交通环境。因此，引入了粒子群优化算法，对这些阈值进行实时的自适应调整。结合自适应调节的阈值，报警函数 Alert(t)能够灵活应对交通流量和密度的变化。尤其是在复杂、动态的交通环境中，通过自

适应调节的阈值设置，模型能够更早、更准确地发出预警信号，帮助交通管理部门和司机提前采取措施，缓解可能的拥堵。

4.5 模型有效性验证

4.5.1 数据集的划分

四个检测点视频数据集：数据集来自四个检测点的视频监控数据，这些数据用于构建和验证模型，确保模型能够捕捉到真实的交通流量变化。

训练集：视频数据被划分为训练集，用于训练时间序列模型或 LSTM 模型。训练集的目的是让模型学习历史数据中的模式和规律。

预测集：预测集则用于模型验证，通过将预测结果与实际观测结果进行对比，来评估模型的准确性。

4.5.2 模型有效性分析：准确率、召回率和调和均值

时间序列/LSTM 模型：利用训练集的数据，时间序列或 LSTM 模型被训练以预测未来的交通流量、速度、密度等参数。

模型验证：在完成训练后，模型会使用预测集进行验证。评估模型在新数据上的表现，确保模型不仅在训练数据上表现良好。

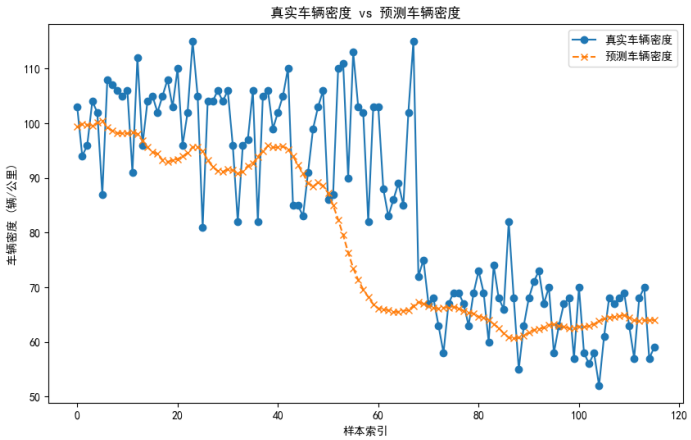

图 4-19　车辆密度的验证

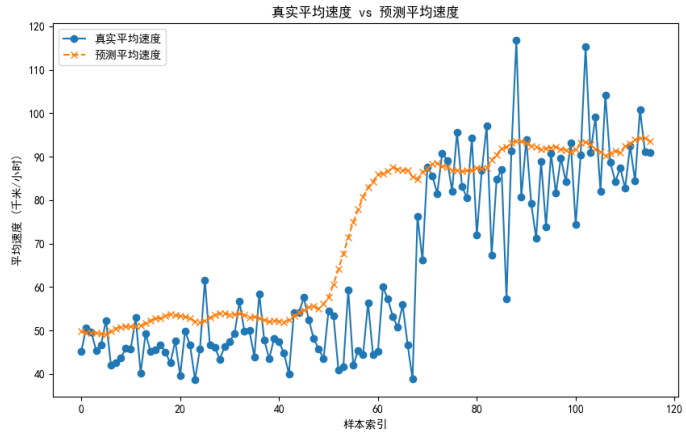

图 4-20　车辆密度的验证

准确率（Precision）：指模型预测为拥堵的时刻中，实际发生拥堵的比例。从图 4-19 和图 4-20 中可以看出，模型预测的车辆密度和速度与实际值较为吻合，尤其是在关键的密度和速度大幅变化的时段，模型能够准确捕捉到趋势。这说明模型具有较高的准确率，能够在预测拥堵时有效避免误报，即减少了预测为拥堵但实际上没有拥堵的情况。

召回率（Recall）：指实际发生拥堵的时刻中，模型正确预测为拥堵的比例。图中显示的趋势一致性表明，模型在拥堵发生时能够及时作出反应，尤其是在车辆密度急剧下降和速度急剧变化的时段，模型的预测与实际情况紧密相关。这意味着模型在检测拥堵发生方面具有较高的敏感性和召回率，能够有效避免漏报，即减少了实际发生拥堵但模型未能预测的情况。

调和均值（F1 Score）：是准确率和召回率的调和平均值，用于综合评估模型的精度和敏感性。结合准确率和召回率的分析，模型在拥堵预测中的表现既准确又敏感，能够较好地平衡误报与漏报之间的关系。调和均值的表现进一步证明了模型在实际应用中的有效性，能够在实际交通状况下提供可靠的拥堵预警。[8]

4.5.3 阈值的有效性

在传统的交通拥堵预测模型中，通常采用固定阈值来判断是否触发拥堵预警。然而，由于道路环境和交通流量的动态变化，固定阈值在长期使用中往往难以保持其有效性。随着时间的推移，这种方法可能会导致检测准确率的下降。因此，为了应对这一问题，我们引入了粒子群优化算法，旨在实现阈值的实时自适应调整，确保模型在不同交通环境下都能保持高效的预测能力。

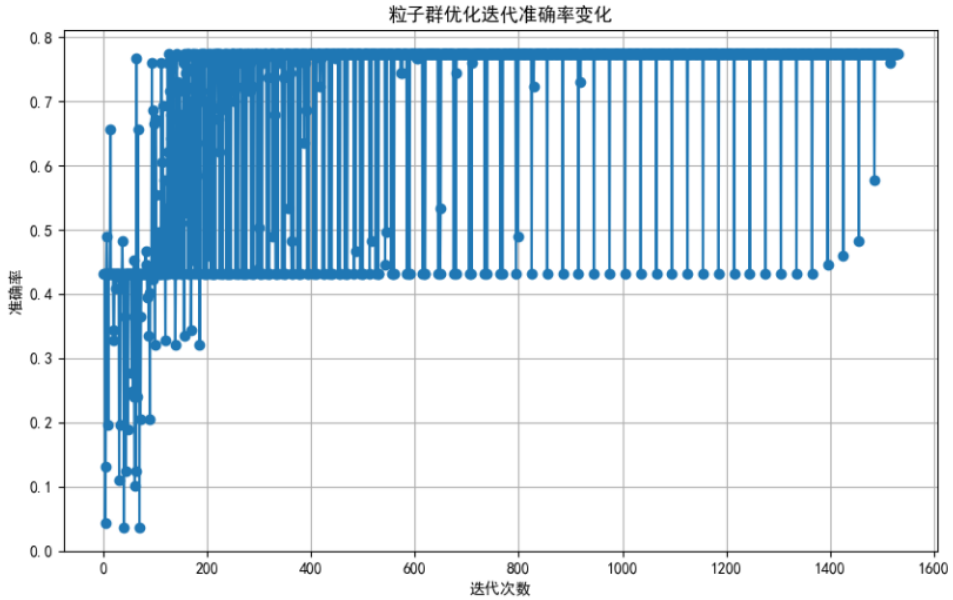

图 4-21　粒子群优化阈值迭代准确率变化

从粒子群优化的迭代过程来看，最初的几次迭代中，模型的准确率波动较大。这表明粒子群在初期阶段对阈值进行了较为广泛的探索，以适应复杂多变的交通环境。随着迭代的深入，准确率逐渐趋于稳定，表明模型已经找到了一个较为合适的阈值，此时进一步的调整对提升检测性能影响不大。最终，准确率在一个较高的水平上保持稳定，验证了粒子群优化在阈值自适应调整方面的有效性。

4.6 小结

在本文中，针对交通流拥堵的实时预警和应急车道启用决策问题，提出了基于视频数据的多目标优化模型，成功构建了交通流拥堵预测与预警系统，并通过多种算法进行优化和验证。

数据预处理与模型建立：

通过对四个检测点的视频数据进行处理，提取了每帧中的车辆数量、速度和密度等交通参数。采用 YOLO 目标检测算法结合多目标跟踪算法，从视频中识别出车辆并进行轨迹跟踪和速度计算。通过这些处理步骤，得到了高质量的数据集，随后以不同的时间间隔进行采样，分别为每 10 分钟和每 2 分钟的粒度，以便捕捉短期与长期的交通流量波动。

传统交通拥堵模型：

首先基于交通流量、速度和车辆密度等基础交通参数，利用经典的交通拥堵模型分析了拥堵状况。通过拥堵度指标（DC）和出行时间比（TPI），成功对不同路段的拥堵等级进行了分类与评估，得出了初始的交通流拥堵阈值。这些阈值为后续模型构建和决策提供了理论依据。

基于时间序列和 LSTM 的预测模型：

为了更精确地预测交通流量和拥堵状况，本文采用了时间序列分析和 LSTM（长短期记忆网络）模型，分别对交通流量、车辆密度和速度等参数进行了短期预测。时间序列模型在分析交通数据的周期性与趋势性方面表现良好，而 LSTM 模型则能更好地捕捉交通流中的复杂非线性动态变化，尤其适合处理长期依赖的时序数据。

粒子群优化阈值自适应调节：

为了提高系统的灵活性和适应性，引入了粒子群优化算法（PSO）对交通流预警模型的阈值进行了自适应调整。该算法有效解决了在动态交通环境中，固定阈值难以适应的问题。通过实时调整交通流量、车辆密度和速度的阈值，系统能够更灵活地应对不同的交通状况，及时预警潜在的拥堵风险。

实时预警系统的设计与验证：

构建了实时预警系统，并结合历史交通数据和实时数据，对四个检测点的数据进行了验证。通过对比历史数据与模型预测值，系统展现了较高的准确率和召回率，尤其在车辆密度和速度急剧变化的时段，模型的预警效果尤为显著。通过与实际交通情况的对比，模型成功验证了其在实际应用中的有效性。

模型的实际应用与优化：

本文提出的模型结合了传统的交通流理论与现代深度学习技术，能够实时分析高速公路上的交通状况，并根据交通流量和车辆密度的变化动态判断是否启用应急车道。通过粒子群优化算法的加持，模型在多种复杂交通场景下表现良好，能够为交通管理部门提供科学的决策依据，减少道路拥堵，提高通行效率。

五、 问题二

5.1 问题分析

构建了一个基于分类算法的决策模型，用于判断何时启用应急车道，以缓解交通拥堵。该模型根据交通流参数，如流量、车速、密度等，进行实时分析，并通过交通流中断概率来确定阈值，以及持续拥挤时间的阈值，为启用应急车道提供决策模型。

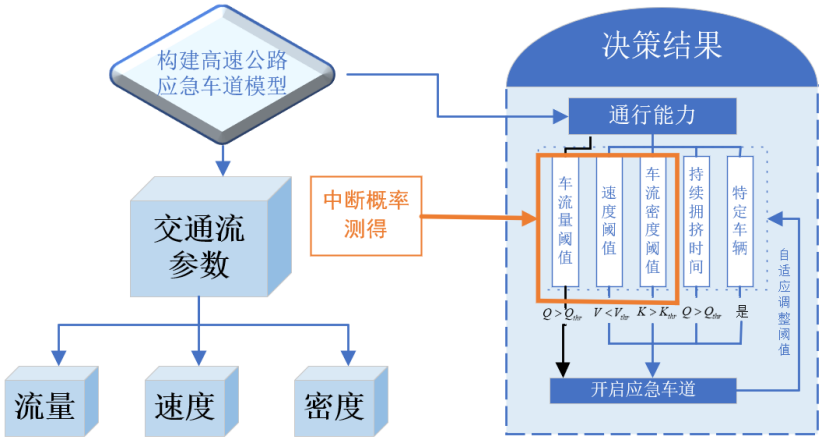

图 5-1　问题二的技术路线图

5.2 应急车道模型参数的理论依据

在道路上移动的车辆展现出了类似流体的动态特性和属性，这些车辆和行人的集合在道路上统称为交通流。本文中所讨论的交通流特指在道路上车辆的持续流动所形成的连续性车流动态，主要聚焦于宏观层面的参数分析，其中三个关键的宏观参数包括：交通流量 q、密度 k（亦称为饱和度）及速度 v，合称为交通流的三大基础参数。

在交通理论实际应用中，不考虑位置和时间因素的影响时，交通流量、速度和密度之间存在基本关系，即

$$q = vk \tag{5-1}$$

定义交通流量 q：该指数描述了在一定时段内经过城市道路某个断面的道路交通实体量，通常以单位时间（小时）内通过的车辆数来衡量，单位为辆/小时。

车速分为两主要类型。地点平均车速指的是在特定时刻通过某一断面的所有车辆速度的算术平均值。区间平均车速涉及在特定时间点，沿一定长度道路行驶的所有车辆的速度分布的平均值。

车流密度 k 反映道路上交通密集程度的关键指标。为确保数据比较的一致性，车流密度通常按照每条车道单独计算，表示为特定时间段内通过特定路段的车辆数除以该车道的长度。

车流密度 k 在特定瞬间，单位道路长度上的车辆数目。该指标的计算可根据为 2-2 所示：

$$k = \frac{N}{L} \tag{5-2}$$

公式中 L 是路段长度， N 是路段内的车辆数。

因车流密度反应的是车道的拥挤程度，所以采用车流密度作为衡量标准去反应不同拥挤程度的信号灯的优化配时。

道路通行能力指的是在一定条件下，单位时间内道路上能够通过的最大车辆数量，“辆/小时”。

$$C = \frac{3600 \times n \times f_w \times f_h \times f_g \times f_p \times f_a}{h} \tag{5-3}$$

n 是车道数量， f_w, f_h, f_g, f_p, f_a 分别为与道路宽度、道路几何形状（如坡度）、信号控制、出入口、天气等因素相关的修正系数， h 是车头时距。

5.3 决策阈值的理论依据

为了评估交通流在下一时间间隔内出现中断（即交通拥堵）的概率，本文采用了基于当前时间间隔中区段车流流速和交通量状况的数据分析方法。Brilon 等人[9]基于细化的交通流量数据，构建了交通流中断概率模型。该模型的主要公式如下：

$$F_c(q) = P(c \leqslant q) = 1 - \prod_{i:q_i \leqslant q} \frac{k_i - d_i}{k_i}, i \in B \tag{5-4}$$

- q_n：每小时每车道的车辆通过量，表示车辆数量，"h" 表示小时，"ln" 表示车道数量。
- c：容量，即在没有交通中断的情况下，一小时内每车道可以通过的最大车辆数量。
- $P(c \leq q)$：表示在时间间隔内，观测到的交通流量 q 超过或等于容量 c 的概率，意味着可能发生交通流中断的概率。
- $F_c(q)$：表示在观测到的交通流量 q 下交通中断的概率。
- q_i：在第 i 个时间间隔内观测到的交通流量。
- k_i：表示交通流量 q 到达 c 时的时间间隔数。
- d_i：表示在第 i 个时间间隔内观察到的、导致速度下降的流量。
- $i \in$B：表示在 B 集合中第 i 个时间间隔的索引，B 是所有使 $q_i \leq q$ 的时间间隔的集合。

为了对拥堵概率分析进行参数估计，模型还引入了统计密度函数 $f_c(q_i)$ 和累积分布函数 $F_c(q_i)$，以通过最大似然技术来估计分布参数。类似的分析函数 L 如下：

$$L = \prod_{i=1}^{n} \left\{ f_c(q_i)^{\delta_i} \cdot [1 - F_c(q_i)]^{\delta_i} \right\} \tag{5-5}$$

其中， δ_i 表示时间间隔内包含未审查对象时 δ_i=1，不包含时 $\delta_i = 0$。

通过该概率模型，可以有效预测下一时间间隔内交通流出现中断的概率，从而测定该道路下速度阈值，流量阈值，密度阈值。

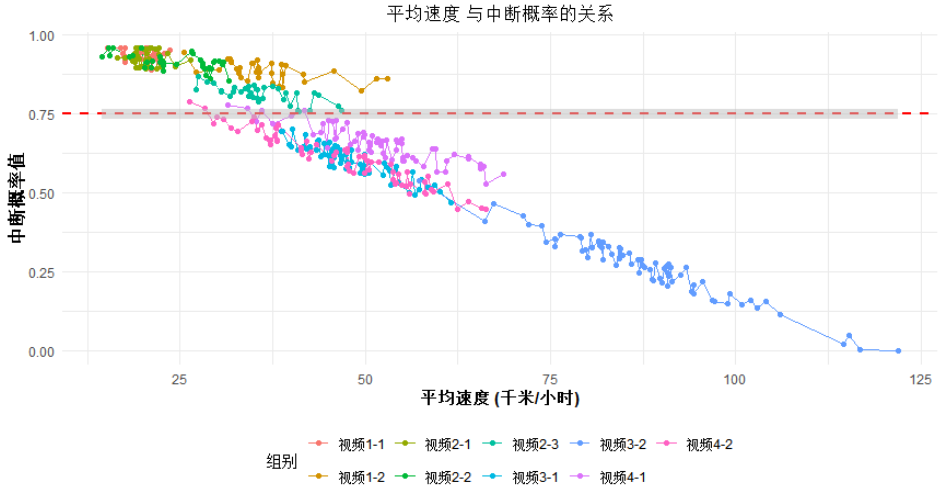

图 5- 2　平均速度与中断概率的关系

由上图可知：当平均速度较高时（大于 75km/h），交通流中断概率较低，基本维持在 0.5 以下。这表明在高速行驶情况下，道路通行较为顺畅，出现拥堵的概率较低。随着平均速度的降低，交通流中断概率呈现上升趋势，尤其在速度降低到 30km/h 以下时，中断概率显著增加，接近甚至超过了 0.75。

根据图像分析，设定的速度阈值为 30km/h 和拥挤概率阈值为 0.75。在这两个阈值的情况下，系统能够有效识别出道路即将进入拥堵状态，并作出相应的预警决策，从而有助于管理和缓解交通拥堵。

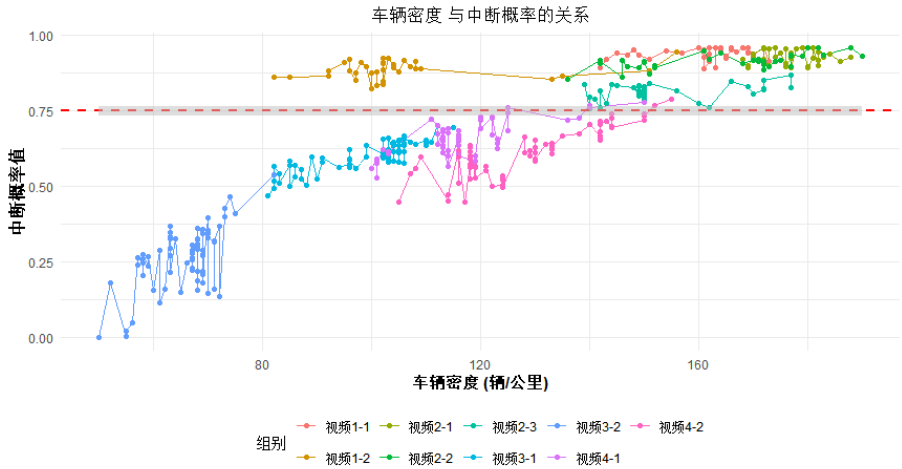

图 5-3　车辆密度与中断概率的关系

图中展示了不同视频组别下，车辆密度与交通流中断概率之间的关系，当车辆密度较低时（小于 100 辆/千米），中断概率一般保持在较低水平，通常低于 0.5。这表明在车辆密度较低的情况下，道路的通行能力较好，拥堵的风险较小。随着车辆密度的增加，中断概率逐渐上升，并在密度接近 140 辆/千米时，超过了 0.75 的阈值。

据图像分析，设定的车辆密度阈值为 140 辆/千米和拥挤概率阈值为 0.75，意味着该道

路段已经接近拥堵状态。

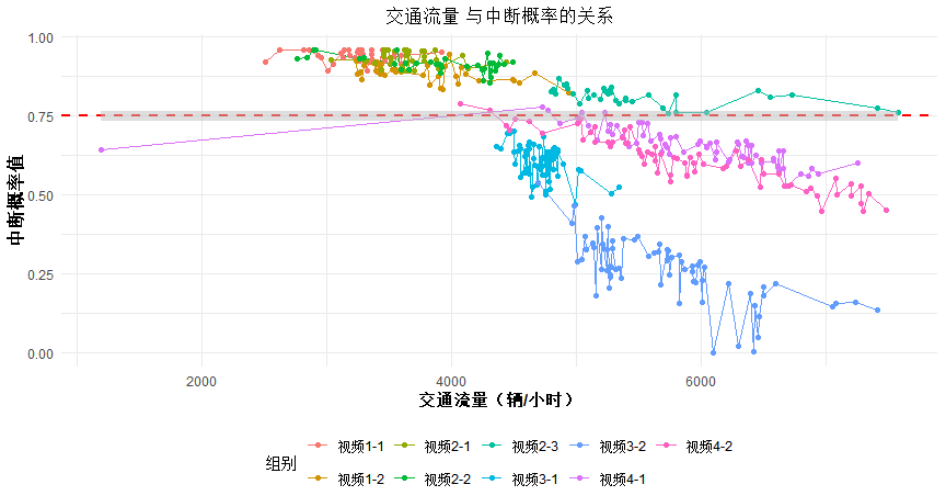

图 5- 4　交通流量与中断概率的关系

图中展示了不同视频组别下，交通流量与交通流中断概率之间的关系。当交通流量较低时（小于 4000 辆/小时），中断概率通常较低，基本维持在 0.5 以下，道路通行顺畅，拥堵概率较低。随着交通流量的增加，中断概率开始上升，并且在流量接近或超过 5000 辆/小时，部分视频组别的中断概率显著超过了 0.75。

这表明在流量达到 5000 辆/小时的情况下，道路已经进入了高拥堵风险区域，所以设定交通流量阈值为 5000 辆/小时和拥挤概率阈值为 0.75。

拥堵等级阈值的确定：用于描述特定路段在某一时间段内维持拥堵状态。为了更加准确地预测和管理交通拥堵，结合前文 *TPI* 得到的拥挤程度来确定。

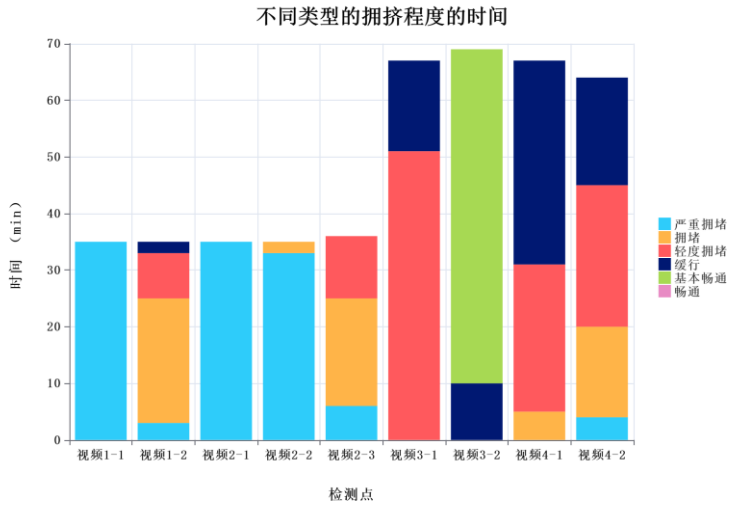

图 5-5 不同检测点拥挤状态的时间

图 5-5 显示了各个拥堵等级（畅通、基本畅通、缓行、轻度拥堵、拥堵、严重拥堵）在不同检测点的时间分布。从图中可以看到，只有在“拥堵”或“严重拥堵”状态下，各检测点的时间占比较大。这意味着这些时段内道路的通行能力已经显著下降，可能导致交通流动不畅。所以将拥堵状态至少达到拥堵，才能开启应急车道。

5.4 高速公路启用应急车道决策模型的搭建

5.4.1 基于阈值的启用判定

基于实时交通流量数据和预测模型来构建决策模型，从而判断是否启用应急车道。

交通流量阈值判断：

流量阈值 Q_{max}：当单位时间内通过某一特定点的车辆数量（流量）超过设定的阈值 Q_{max} 时即：

$$Q > Q_{max} \tag{5-6}$$

表明该路段的车辆数量已达到道路通行能力的极限，存在发生交通拥堵的高风险。此时，系统将考虑启用应急车道以分流部分车辆，缓解交通压力。

车速阈值判断：

速度阈值 $V_{\min}$：当平均车速降至低于某个临界值 $V_{\min}$ 时即：

$$V < V_{\min} \tag{5-7}$$

表明交通流正在减速，可能已经开始出现拥堵迹象。低速行驶的车辆往往会导致车流密度增加，因此，如果车速持续低于阈值，系统将判断为需要启用应急车道，以提高道路通行速度。

车流密度阈值判断：

密度阈值 $K_{\max}$：车流密度指单位长度道路上的车辆数量。当车流密度达到或超过设定的阈值 $K_{\max}$ 时，

$$K > K_{max} \tag{5-8}$$

意味着道路上的车辆过于集中，容易引发交通堵塞。在此情况下，启用应急车道可以有效分流车辆，降低密度，缓解拥堵。

拥堵等级判断：

拥堵等级根据交通状况划分为“畅通”、“基本畅通”、“缓行”、“轻度拥堵”、“拥堵”和“严重拥堵”六个级别。为了触发应急车道的启用，要求拥堵等级达到或超过“缓行”状态。交通拥挤程度 $R(t)$ 需要进行编码，1 表示畅通，2 表示基本畅通，3 表示缓行，4 表示轻度拥堵，5 表示拥堵，6 表示严重拥堵。

5.4.2 模型构建

通过历史和实时交通数据，训练一个分类模型来预测是否需要启用应急车道。模型的输入数据是交通流参数（如流量、速度、密度等）以及天气等外部因素，输出是分类结果 $E(t)$，表示是否需要启用应急车道。

$$X = \left[q(t), v(t), k(t), R, \ldots\right], \quad y = E(t) \tag{5-9}$$

约束条件：

$$s.t.\begin{cases} Q > Q_{max} \\ V < V_{\min} \\ K > K_{max} \\ R(t) \geq 4 \end{cases} \tag{5-10}$$

目标变量 $E(t)$是一个二分类标签，表示在时间点 t 是否需要启用应急车道。

$E(t)=1$：表示需要启用应急车道。

$E(t)=0$：表示不需要启用应急车道。

5.4.3 自适应阈值调整

通过实时分析交通数据和历史数据，系统可以动态根据拥堵概率进行动态自适应更新，以确保系统能够更加准确地反映当前的道路状况并做出及时的决策。

系统通过部署在道路上的传感器、摄像头和其他监控设备，持续获取实时的交通流参数（如流量、车速、车流密度等）。这些实时交通数据的输入计算拥堵概率，系统会根据当前检测到的拥堵概率来调整阈值。[10]

更新策略：

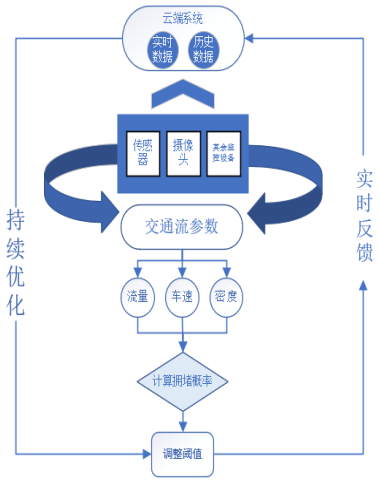

图 5-6　自适应阈值调整流程图

拥堵概率驱动：每当系统检测到新的交通数据时，都会计算当前的拥堵概率。如果该概率在一定范围内波动，则阈值保持不变；如果拥堵概率显著增加或减少，则系统会根据这个变化调整阈值。

反馈机制：通过对过往决策效果的分析（如是否有效避免了拥堵），系统将对阈值调整策略进行持续优化。例如，如果发现当前阈值设定导致频繁触发应急车道但并未显著缓解拥堵，系统将自动调整阈值，以避免不必要的启用。

应用效果：通过这种基于拥堵概率的自适应阈值调整，系统能够在不同的交通状况下，灵活地调整流量、速度和密度的阈值，确保在可能出现拥堵时提前做出反应，并有效利用应急车道来缓解交通压力。

5.5 小结

本次研究主要针对高速公路应急车道的启用问题。

采用了一种基于交通流参数的中断概率模型。该模型通过当前时间间隔内的车流流速和交通量状况来计算中断概率，并基于这些数据得到已经范围的交通流参数的阈值。

根据实时交通数据，将道路的拥堵状态划分为六个级别，并在拥堵程度达到或超过“缓行”状态时触发应急车道的启用。

基于阈值的模型：设定车流量、速度、密度等相关阈值，提出了一个基于实时交通数据的应急车道启用决策模型。

自适应阈值调整：系统通过实时交通数据进行拥堵概率的动态计算，并自动调整阈值，确保在不同时段能够及时有效地启用应急车道。

研通过构建实时交通参数的优化模型和阈值判断机制，系统可以在道路面临拥堵高风险时及时做出决策，启用应急车道从而分流车辆，降低拥堵程度。自适应阈值调整策略进一步确保了系统能够根据历史数据反馈，不断优化决策准确性，减少应急车道的误用或滥用，提升交通管理效率。

六、 问题三

6.1 问题分析

实时算法的构建可以结合问题一的预测模型，预测模型将利用历史数据和当前实时数据对未来几分钟或更长时间的交通状况进行预测。将预测出的交通参数作为输入，输入到问题二中构建的应急车道开启决策模型。同时可以利用常见的机器学习分类算法，进行分类，多种算法结果对比，寻找适合当前交通状况的最优模型。量化应急车道开启前后的交通变化，可以收集实际交通流量、速度、密度等参数的变化，进行详细对比分析。利用图表展示开启应急车道前后的交通参数变化，如流量变化图、速度变化图、拥堵时长对比图等，直观展示应急车道对交通流改善的效果。

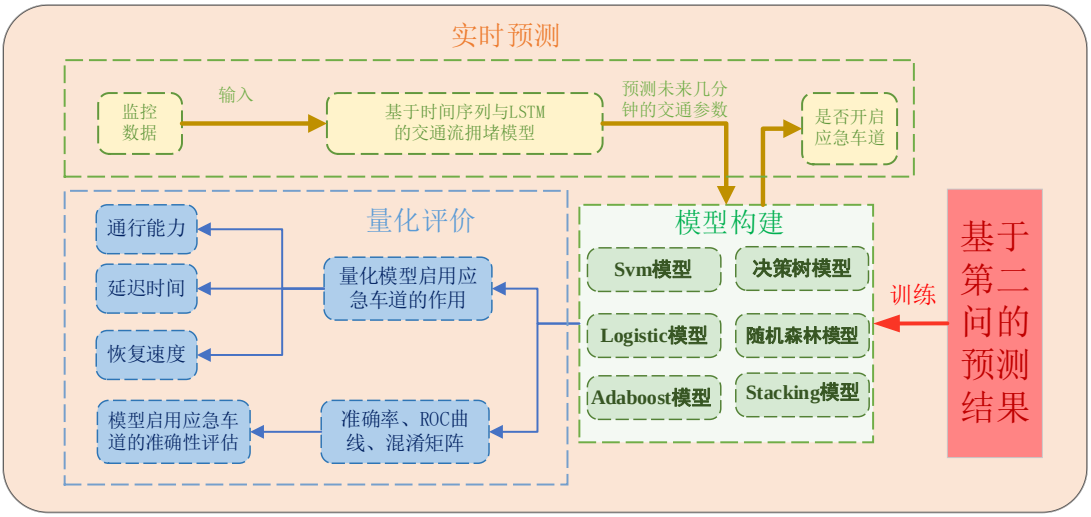

图 6-1　问题三的技术路线图

6.2　实时预测模型的构建

在实时决策是否启用应急车道的过程中，利用了第一问中构建的交通流拥堵模型，通过该模型预测未来几分钟内的交通参数，从而做到实时预测。这一过程的核心在于基于时间序列与 LSTM（长短期记忆网络）模型，对交通流量、车速、密度等关键交通参数进行短期预测，以便及时判断是否需启用应急车道。

监控数据输入：实时收集的交通监控数据（如流量、速度、车流密度等）被输入到构建的交通流拥堵预测模型中。

使用时间序列与 LSTM 模型对未来几分钟的交通参数进行预测，LSTM 的优势在于它能够很好地捕捉数据中的长期依赖关系，从而为交通拥堵情况的短期预测提供准确的结果。

根据预测得到的交通参数，应用之前设定的启用应急车道的阈值规则（例如流量、速度、密度超过阈值，或者拥堵等级达到一定程度），系统会自动判断是否需要启用应急车道。

6.3　基于规则的应急车道启用模型的求解

基于第二问所建立的模型与启用规则：

$$X=\left[q(t),v(t),k(t),R,\ldots\right],\quad y=E(t) \tag{6-1}$$

$$s.t.\begin{cases} Q>Q_{max} \\ V<V_{\min} \\ K>K_{max} \\ R(t)\geq 4 \end{cases}$$

输入数据：

每个数据样本代表一个时间点的交通状态，采样频率为每 2 分钟。输入特征包括：

- 流量 q(t)：每小时通过某观测点的车辆数（辆/小时）。
- 速度 v(t)：车辆在某观测点的平均速度（千米/小时）。
- 密度 k(t)：每公里道路上车辆的数量（辆/公里）。
- 交通拥挤程度 R(t)：5 表示拥堵，6 表示严重拥堵。

模型求解结果：

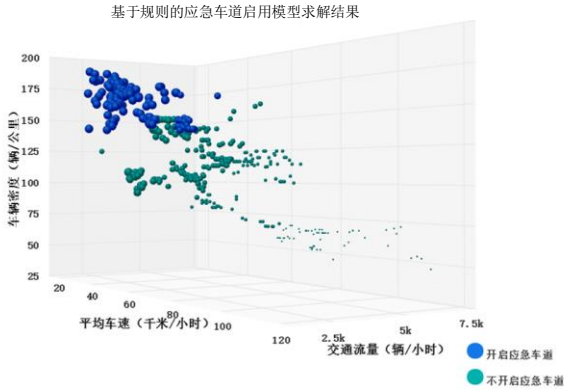

图 6-2　基于规则的应急车道启用模型求解结果

图 6-2 展示了基于规则的应急车道开启模型的 3D 可视化结果，其中三个轴分别表示交通流量（辆/小时）、平均车速（千米/小时）、车辆密度（辆/公里）。数据点的颜色区分了在不同交通条件下是否开启应急车道：蓝色点表示开启应急车道，绿色点表示未开启应急车道。

可以形象的看到，当交通流量增加、平均车速降低、车辆密度升高时，更容易触发应急车道的开启。该图以直观地理解规则模型的运作，并展示不同交通状况下的决策结果。

该模型的优势在于它基于明确的阈值和规则，具有较强的可解释性；但同时，模型的灵活性可能不足，对于一些复杂的交通情况可能无法及时做出最优决策。因此，在下文团队结合其他机器学习进行优化和补充。

6.4　建立基于规则的机器学习应急车道开启模型

在前面的模型中，建立了基于交通流量、速度、车流密度等参数以及设定的阈值，通过逻辑判断来决定是否启用应急车道。这种基于规则的决策模型简单直观，但其效果依赖于阈值的选择，并且在面对复杂的交通状况时，可能无法有效适应变化的环境。

为了进一步提升模型的灵活性和精确性，引入了机器学习分类算法。与基于阈值的模型不同，机器学习算法通过学习历史数据中的模式和关系，自动提取特征并做出更加精准

的决策。可以考虑更多的影响因素，能自适应地调整决策规则，从而提高模型在不同交通状况下的适用性和可靠性。

本小结将使用多种常见的机器学习分类算法，包括支持向量机 (SVM)、随机森林、决策树、AdaBoost、BP 神经网络和 Stacking 集成模型等，对交通数据进行分类训练，并通过对比不同算法的性能，选出最适合实际应用的模型，用于应急车道的启用决策。

各个机器学习分类算法的优劣性如表 6-1 所示：

表 6-1　机器学习分类算法的优缺点

模型名称	适用情况	优势
SVM	数据维度较高，且样本数量相对较少的情况下效果较好。	对高维数据有良好的分类效果。 对边界样本有良好区分能力。
决策树	数据量中等、特征较多且需要快速构建模型使用。	可解释性强，结果易于理解。 训练速度快。
随机森林	数据维度较高且噪声较多的情况下效果较好，适合应对复杂问题。	抗噪能力强，能够处理高维数据 防止过拟合，模型稳定性高。
AdaBoost	适用于需要提升弱学习器效果的情况，特别是处理中小型数据集时。	可有效提升弱学习器性能。 处理噪声较少的情况表现出色。
BP 神经网络	适用于大规模数据，非线性关系较复杂的情况，特别是深度学习场景。	适合处理复杂的非线性问题。 可以处理大量数据，模型灵活度高。
Stacking 集成模型	适合需要集成多个模型以提升预测性能的情况，常用于模型对比实验。	能够结合多种模型的优势，提升准确率。 适用于各类数据类型和特征集。

6.4.1 基于 SVM（支持向量机）的分类模型

SVM 的目标是在样本数据的不同类别之间找到一个能够最大化两类数据之间边界的超平面，用来分类是否需要启用应急车道，即通过对交通参数的综合分析，判断当前交通状态是否满足启用应急车道的条件。

决策函数：

$$f(x) = \text{sign}(\langle w, x \rangle + b) \tag{6-2}$$

其中，w 是权重向量，代表各个交通参数（如流量、速度、密度等）的权重。

x 是输入向量，包含实时监测到的交通数据。

b 是偏置，决定了决策边界的平移。通过 w 和 b 的优化，SVM 能够找到一个最佳的超平面，用于将需要启用应急车道的交通状态（正类）与不需要启用的状态（负类）区分开来。

优化问题：

$$\min_{w,b} \frac{1}{2}\| w\|^2 \tag{6-3}$$

约束条件：

$$y_i(\langle w, x_i\rangle + b) \geq 1, \quad i = 1, 2, \ldots, n \tag{6-4}$$

其中， y_i 是样本的标签，+1 表示需要启用应急车道，−1 表示不需要启用应急车道。x_i 是样本点，包含实际采集到的交通参数。

拉格朗日函数：

$$L(w, b, \alpha) = \frac{1}{2}\| w\|^2 - \sum_{i=1}^{n} \alpha_i [y_i(\langle w, x_i\rangle + b) - 1] \tag{6-5}$$

α_i 是拉格朗日乘子，用于将约束条件引入优化目标。通过优化拉格朗日函数，SVM 模型最终能够找到一个最优解，用于判定当前交通状态是否满足启用应急车道的条件。

6.4.2 基于决策树的分类模型

决策树的目标是通过不断选择最佳特征来将数据集划分为纯度尽可能高的子集，从而实现对目标变量的分类。在应急车道启用模型中，决策树可以通过选择不同的交通参数（如流量、速度、密度等）来决定是否应启用应急车道。[11]

信息增益公式：

决策树的核心在于通过选择能够最大化信息增益的特征来构建分支。

$$IG(D, A) = H(D) - \sum_{v \in Values(A)} \frac{|D_v|}{|D|} H(D_v) \tag{6-6}$$

$IG(D, A)$ 是特征 A 对数据集 D 的信息增益。A 对应于交通流参数中的某一个，例如流量、速度、密度。D 是实时监控的交通数据集。$H(D)$ 是数据集的熵，表示数据集的纯度。计算公式为：

$$H(D) = -\sum_{i=1}^{k} p_i \log_2 p_i \tag{6-7}$$

p_i 表示某个特定交通状态（如“拥堵”、“畅通”）的发生概率。

通过信息增益最大化，决策树可以逐步分裂数据集，从而构建出用于分类的决策路径。在每个叶节点，决策树通过检查交通参数是否满足特定条件来最终决定是否启用应急车道。

6.4.3 基于随机森林的分类模型

通过结合多个决策树的预测结果来提高模型的准确性和鲁棒性。在交通拥堵预警和应急车道启用中，随机森林能够有效地处理多维度的交通流量数据，提供更为可靠的决策支持。

随机森林的基本原理：

特征选择：在构建决策树的每个节点时，从总特征集中随机选择一个特征子集F用于节点分裂。

节点分裂：计算信息增益或基尼系数，选择能够最大化纯度提升的特征及其对应的分裂点进行节点分裂。

投票：假设有 N 棵树，每棵树独立地预测交通状态（是否需要启用应急车道），随机森林将通过多数投票决定最终结果

$$E(t) = \text{MajorityVote}(E_1(t), E_2(t), \ldots, E_N(t)) \tag{6-8}$$

$E_i(t)$ 是第 i 棵数的预测结果，即是否启用应急车道的决策。

通过随机森林的集成方法，模型能够综合考虑不同时间点和条件下的交通数据，给出较为可靠的决策建议，用于指导是否需要启用应急车道以缓解交通压力。

6.4.4 基于 AdaBoost 的分类模型

AdaBoost（Adaptive Boosting）是一种基于“加法模型”和“前向逐步加法算法”的提升方法。其核心思想是通过组合多个弱分类器，来构建一个强分类器。在每一轮迭代中，AdaBoost 会根据前一轮分类器的表现调整数据的权重，使得那些被错分的样本在下一轮训练中获得更多的关注。最终，AdaBoost 将所有弱分类器的加权结果综合，形成最终的分类决策。

在交通拥堵预警和应急车道启用模型中，AdaBoost 通过组合多个简单的分类器，基于流量、速度、密度等单一特征的决策树，来构建一个能够更精确判断交通状态的强分类器。

$$w_1(i) = \frac{1}{m}, \quad i = 1, 2, \ldots, m \tag{6-9}$$

m 是样本总数，各个时间点的交通参数，包括流量、速度、密度等，每个样本初始权重相等。

训练弱分类器：对于每一轮 $t = 1, 2, \ldots, T$

使用当前权重 w_t 训练弱分类器 $h_t(x)$

计算分类误差：

$$\epsilon_t = \frac{\sum_{i=1}^{m} w_t(i) I(y_i \neq h_t(x_i))}{\sum_{i=1}^{m} w_t(i)} \tag{6-10}$$

最终分类器：

$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right) \tag{6-11}$$

最终分类器结合多个弱分类器的权重，得出应急车道开启与否的最终判断。

6.4.5 基于 BP 神经网络的分类模型

BP（Backpropagation）神经网络是一种多层前馈神经网络，其通过反向传播算法来训练网络。BP 神经网络由输入层、隐藏层和输出层组成，输入层接受特征输入，隐藏层通过非线性激活函数处理信息，输出层给出预测结果。反向传播算法通过计算损失函数的梯度，逐步调整网络权重，使模型预测更加准确。

在交通拥堵预警和应急车道启用的项目中，BP 神经网络可以处理多维度的交通数据，并捕捉复杂的非线性关系。特别是对于复杂的交通流量模式，BP 神经网络可以通过多层神经元的组合来学习和预测，进而准确判断是否需要启用应急车道。

前向传播：

$$a^{(l)} = f(W^{(l)}a^{(l-1)} + b^{(l)}) \tag{6-12}$$

每一层的输出 $a^{(l)}$ 通过激活函数计算得到的中间结果。这里的输入 $W^{(l)}$ 和偏置 $b^{(l)}$ 对应的是交通特征（流量、速度、密度）以及经过网络每层计算得到的中间激活结果。

损失函数：

$$L = \frac{1}{2m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 \tag{6-13}$$

预测值 $\hat{y}_i$ 是神经网络预测的应急车道开启与否的决策，真实值 y_i 则是根据数据判断的实际应急车道状态。

反向传播：通过反向传播计算每一层的权重更新值，调整神经网络的参数，使得网络能够更好地预测未来时间点应急车道是否应该开启。

权重更新：

$$W^{(l)} = W^{(l)} - \eta \frac{\partial L}{\partial W^{(l)}} \tag{6-14}$$

权重更新的过程就是不断调整模型，使其能更加准确预测基于当前交通状况是否需要开启应急车道。

6.4.6 基于 Stacking 集成模型的分类模型

Stacking 是一种集成学习方法，通过将多个不同类型的基础分类器（如决策树、SVM、神经网络等）的预测结果作为新的输入特征，再通过一个元学习器（meta-learner）来进行最终的预测。与其他集成方法不同，Stacking 能够利用不同模型的优势来提升预测的准确性。[12]

在交通拥堵预警和应急车道启用的项目中，单一的分类器可能无法充分捕捉所有交通数据的特征。Stacking 通过结合多个模型的预测结果，能够提高决策的准确性和鲁棒性，

特别是在面对复杂、多变的交通状况时，能够更好地综合各个模型的优点，给出更为可靠的决策结果。

基础模型：假设有 M 个基础模型，分别为 $f_1(x), f_2(x),\ldots, f_M(x)$，每个模型根据输入 x 给出一个预测结果。这些模型的输入数据 x 代表交通流参数，输出结果是预测的是否需要启用应急车道。

元学习器：将基础模型的输出作为新的输入特征 $Z = f_1(x), f_2(x),\ldots, f_M(x)$，输入到元学习器中进行最终的预测。在交通应急车道启用模型中，元学习器的输入特征是多个 Z 基础模型的预测结果组合，元学习器通过这些组合预测出最终的应急车道启用决策。

模型训练：通过交叉验证的方式，先在训练集上训练各个基础模型，再在它们的预测结果上训练元学习器。对于交通应急车道启用决策来说，先用不同的机器学习模型（基础模型）学习和预测交通参数，然后再通过一个元学习器整合这些模型的预测结果，得出最终的启用决策。

6.4.7 基于规则的机器学习应急车道开启模型的评估

在训练机器学习模型时，发现不进行交叉验证时，各模型的准确率都非常高。然而，这种高准确率可能是由于模型过拟合（Overfitting）所导致的。过拟合意味着模型在训练数据上表现很好，但在新数据上表现不佳。因此，为了验证模型的泛化能力（Generalization Ability）并避免过拟合，采用了交叉验证技术。

为了评估不同机器学习模型的性能，使用了混淆矩阵来展示各模型在交叉验证中的分类表现。

图 6-3 展示了 SVM、决策树、随机森林、AdaBoost、BP 神经网络和 Stacking 模型的混淆矩阵。通过这些混淆矩阵，可以观察到各模型在正类和负类样本上的分类精度。其中，随机森林和 Stacking 模型表现最佳，准确率和召回率均接近完美，表明它们能够很好地识别出应急车道开启的情况。

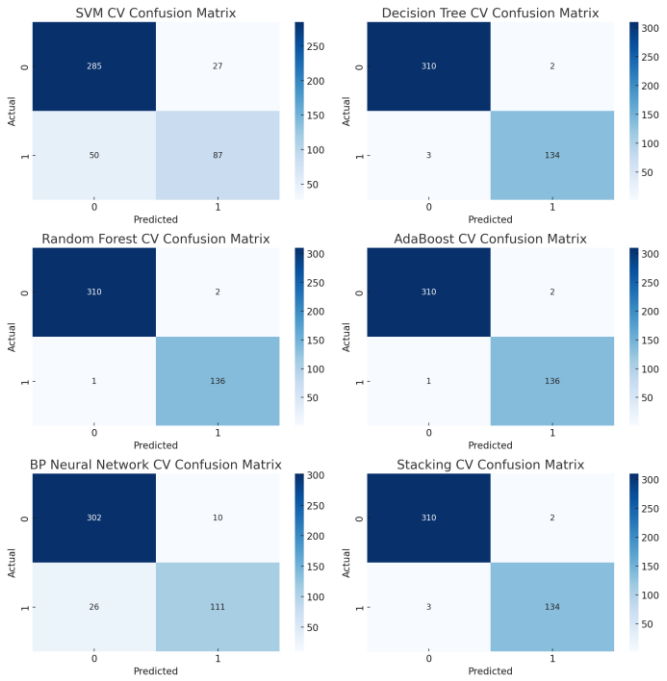

图 6-3　各机器学习模型的交叉验证混淆矩阵

为了进一步比较各模型的整体分类性能，绘制了 ROC 曲线并计算了 AUC 值，如图 6-4 所示。ROC 曲线展示了模型在不同阈值下的表现，其中 AUC 值越接近 1，模型的整体性能越好。从图中可以看到，随机森林、AdaBoost 和 Stacking 模型的 AUC 值均达到 1.0，表现非常优秀，而 SVM 相对表现稍差。

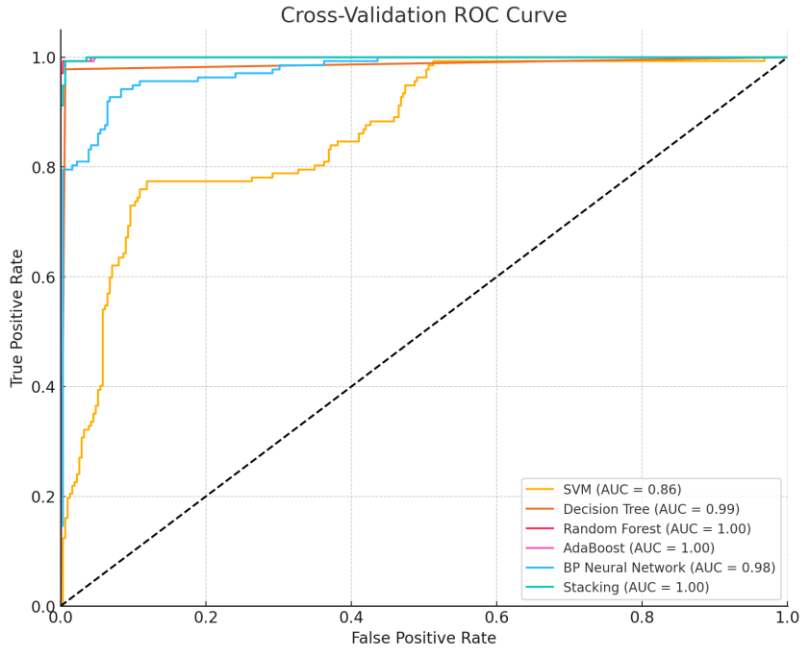

图 6-4　各机器学习模型的 ROC 曲线

在本节中，比较了多种常见的机器学习分类算法的表现，以评估各算法在应急车道启用问题中的适用性。综合对比多个模型后，具体对比结果如表 6-2 所示。

表 6-2　机器学习的结果对比

机器学习类别	机器学习分类的结果对比
SVM	SVM 对本项目表现不佳，误分类率高，特别是假负类多
决策树	决策树非常适合此数据集，大部分情况能很好分离正负类
随机森林	随机森林是表现最好的模型之一，尤其是在多维数据上
AdaBoost	AdaBoost 表现非常优秀，特别在处理样本不平衡时
BP 神经网络	神经网络表现中等，误分类较多，需要更多的训练数据

6.5 基于模型的应急车道启用效益量化分析

6.5.1　时间段的选择依据

根据前面构建的模型（如 LSTM 交通流预测模型），可以预测出未来几分钟后的交通参数变化情况。通过这一预测，能够提前判断哪一段时间将出现交通拥堵，并据此决定是否启用应急车道。应急车道的开启时机直接依据模型的预测结果，确保在最合适的时间点介入，以达到最佳的拥堵缓解效果。

通过模型的预测，选取第三个检测点的视频的第 4min，第 10min，第 52min 时，决策

模型提醒开启应急车道，前方 3min 后发生交通拥堵。通过模型的预测，选取第三个检测点的视频中的第 4 分钟、第 10 分钟、第 52 分钟的关键时刻，决策模型提示需要开启应急车道。在这些时刻，模型预测出前方在大约 3 分钟后会发生交通拥堵，为了避免交通流量的进一步恶化，系统建议提前启用应急车道以分流车流，降低交通密度。

团队选这三个关键节点进行量化应急车道开启效果时，选择了基于车流波的三个关键指标来评估开启应急车道后的改善情况。

6.5.2 车流波理论

在 1955 年，Lighthill 等学者[14]提出了交通流模型，交通流的传播模型采用了车流波理论，该理论通过类比流体动力学原理，将车流视为流体，从而建立了描述车流连续性的方程[17]。这一理论框架助于揭示道路流量、密度与速度之间的相互作用。

假设在一条笔直且长度足够长的道路上，车辆沿路线正方向 x 方向行驶，当前道路上的交通流的处于状态 B，在进入某个瓶颈路段，该路段的车流量激增，交通流的状态由 B 变为 A，其流量、速度、密度也由 q_b、v_b 和 k_b 转变为 q_a、v_a 和 k_a，此时两个交通流状态会形成波面 S，假设改波面的传播速度为 v_s，如图所示。

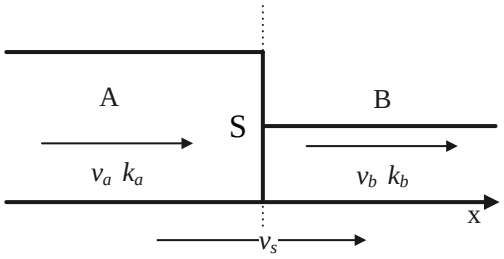

图 6-5　不同密度下的车流运行图

在时间 t 内穿过 S 分界面的车辆数为 N：

$$N = k_a[(v_a - v_s)t] = k_b[(v_b - v_s)t] \tag{6-15}$$

即：

$$k_a(v_a - v_s) = k_b(v_b - v_s) \tag{6-16}$$

$$v_s = \frac{k_a v_a - k_b v_b}{k_a - k_b} \tag{6-17}$$

假设 A、B 的车流量为 q_a、q_b，即：

$$q_a = k_a v_a \quad q_b = k_b v_b \tag{6-18}$$

波速可以写为：

$$v_s = \frac{q_a - q_b}{k_a - k_b} \tag{6-19}$$

假设道路交通流的特征遵循由 GreenShields 等人[13]提出的速度-密度线性关系模型。在这一模型框架下，速度与密度之间存在一种单调递减的线性关联。该模型的主要优势在于其简洁性和直观性，众多研究支持，对于常见的交通流密度水平，此模型与实际观测数据之间有着良好的吻合度。根据该模型，可以推导出速度与密度之间的具体数学关系式。

$$v(k) = v_f(1 - \frac{k}{k_j}) \tag{6-20}$$

此式子中，v_f 是畅行速度，k_j 是阻塞密度。

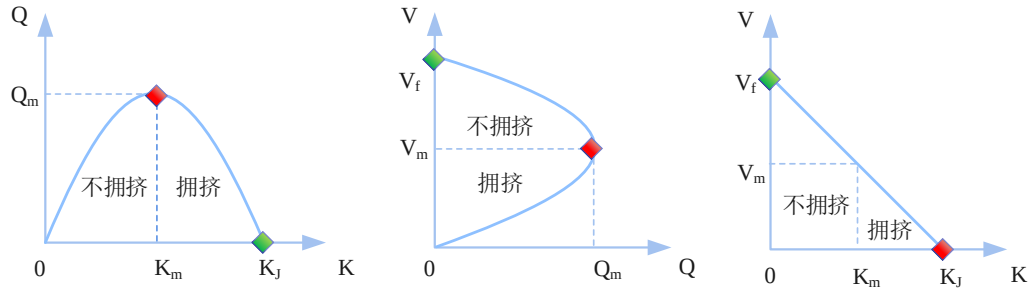

图 6-6　流量-密度-速度关系图

交通流的动态变化在交叉口受到车辆停车与启动的影响，而信号相位的轮换引发了交通流的周期性变化，从而引起交通密度的波动，这些冲击波可细分为排队波和消散波两类。

在红灯期间，排队波的形成导致交通密度增加，车辆速度逐渐减至零，并在交叉口停止线前形成排队。反之，在绿灯期间，消散波的出现促使车队重新启动，车辆速度从零恢复至自由流速度，使车辆得以依次通过交叉口。

排队波：红灯时，车辆在交叉口的停止线前逐渐停下。最先到达此区域的车辆开始减速，随后到达的车辆也相继减速。这一过程导致了车流密度的变化，其中，变化前车流的密度记为 k_a，流量为 q_a。当车流经过冲击波波阵面 S 时，密度变为 k_j，流量减至零。这一冲击波从点 C 开始，逆着车辆行进方向向交叉口上游传播。以 F 为波阵面，排队波的扩散速度为 v_s，方向与车辆行进方向相反，在交叉口停止线前形成车辆排队现象。

消散波：绿灯时，位于停止线前的第一辆车以饱和流率重新启动，后续的车辆也逐渐跟随启动并通过交叉口。车流在经过波阵面 S 后，密度从最大值变化至自由流密度，流量则从零增至 q_b。此时，消散波产生，并以停止线为新的波阵面，向后扩散。消散波的扩散速度为 v_s，其方向与交通流的行进方向一致。

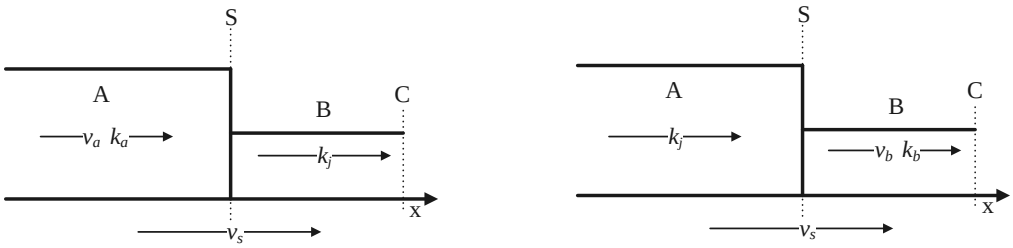

图 6-7　排队波和消散波示意图

6.5.3　基于车流波的三个量化

重要参数的确定：

关于开启应急车道的道路的通行能力变化，参考刘强的研究，应急车道的最大通行能力是 1200 辆/h，所以再开启应急车道以后，三车道的高速公路的最大服务能力为 5600 辆/h。

对于预测时间段，以提前 2min 为预测基准。

消散时间的计算：

$$T = \frac{Q - C_1}{C_{\text{total}} - Q} \tag{6-21}$$

$Q - C_1$表示队列积累的速率

$C_{\text{total}} - Q$表示队列消散的速率

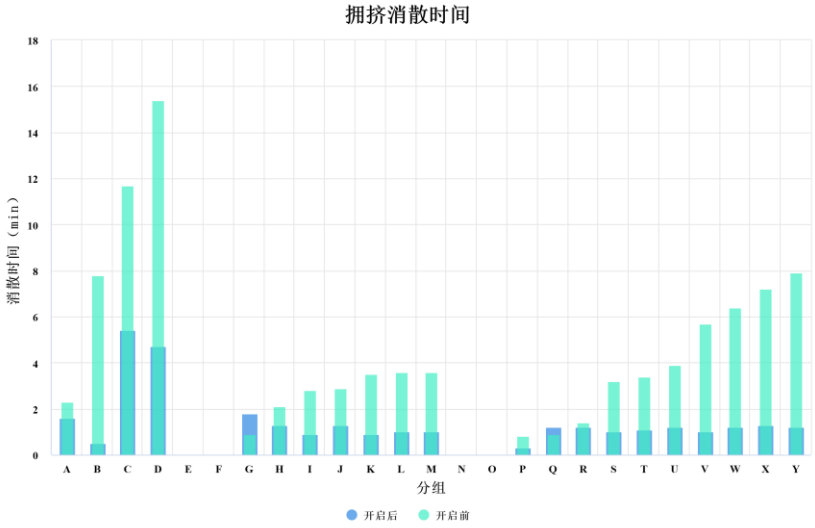

图 6-8　拥挤消散时间对比

在未开启应急车道的情况下，消散时间在某些分组显示出明显的高值，消散时间积累较多。

因为拥堵未能有效缓解，车辆通行量继续超出道路通行能力，导致队列不断积压，时间不断延长，消散时间显著增加。

开启应急车道后，消散时间在每个分组都显著降低。在一些组（中，开启应急车道后消散时间明显较短，未出现时间累积现象。

这表明，在开启应急车道后，额外的通行能力提升缓解了交通压力，拥堵迅速消散，从而消除了消散时间的积累现象。

消散长度的计算：

$$L_b = w_b \times T_b = \frac{Q_2 - Q_1}{k_2 - k_1} \cdot T_b \tag{6-22}$$

L_b 是消散长度。

w_b 是波速，及车流波的传播速度。

T_b 是消散时间。

Q_2 和 Q_1 是不同状态下的车流量。

$k_2 k_1$ 是不同状态下的交通密度。

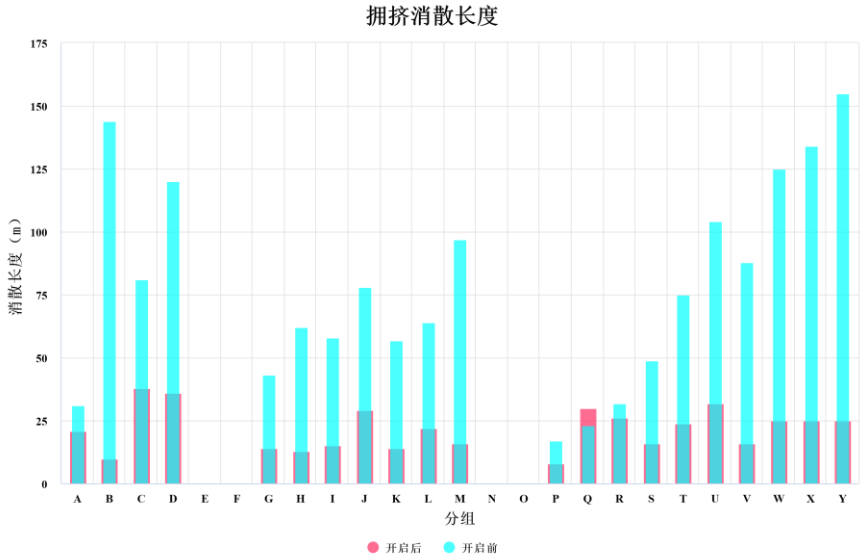

图 6-9　拥挤消散长度对比

未开启应急车道时的消散长度（以浅蓝色柱状表示）在许多分组中显示为较大的数值，特别是在 D、F、Y 等组，消散长度达到了 100 米以上，甚至接近 150 米。

开启应急车道后（以粉色或灰色柱状表示），消散长度显著减少，尤其是在高流量分组（例如 D、F、Y 组），消散长度从 100 米以上降低到 50 米左右，甚至更低。

在大多数情况下，开启应急车道显著减少了拥堵的消散长度，尤其是在高流量组中应急车道的效果尤为明显。通过这些数据，交通管理者可以更好地判断应急车道在不同路段的有效性，并决定在何时、何地开启应急车道以缓解交通压力。

消散车辆数：

$$N_b = Q_w \cdot T_b = \frac{V_2 - V_1}{\frac{1}{k_2} - \frac{1}{k_1}} \cdot T_b \tag{6-23}$$

N_b：消散车辆数。

Q_w：消散波流量。

T_b：消散时间。

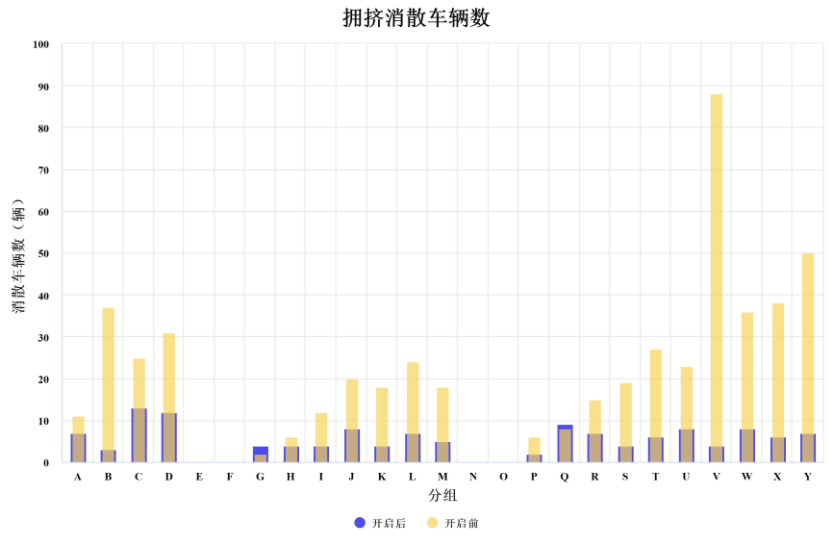

图 6-10　拥挤消散车辆数对比

未开启应急车道时（以黄色柱状表示），在许多分组中显示出较高的车辆消散数量，特别是在 Y、W、T 组，车辆数达到 70 辆甚至更高。这表明，在这些分组中，未开启应急车道时，车辆的排队现象较为严重，造成了较大的积压，需要消散的车辆数量较多。

在开启应急车道后（以蓝色柱状表示），消散车辆数量明显减少，尤其在高流量分组（如 Y、T、W）中，车辆数量从高峰值大幅下降至不到 20 辆，甚至更少。这表明，开启应急车道后有效减少了交通流量的积压，缩短了拥堵的消散时间，并显著降低了车辆数量。

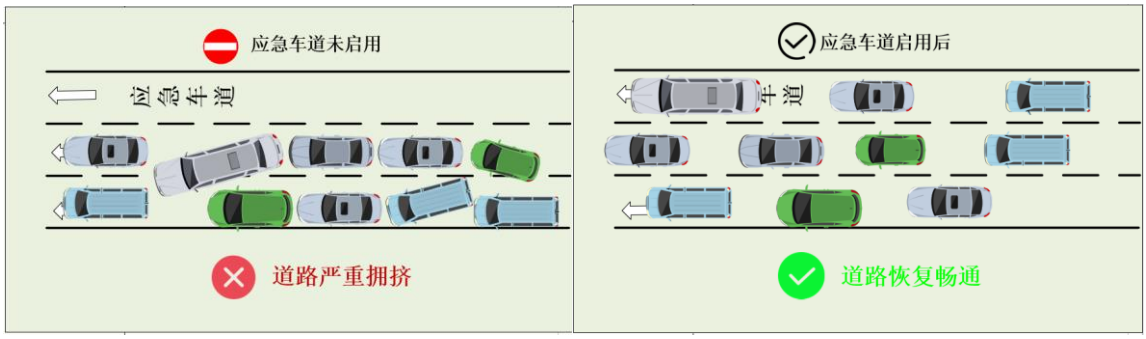

图 6-11　应急车道启用后的对比图

6.6　小结

在应急车道开启后，基于车流波理论的三个量化指标（消散时间、消散长度、消散车

辆数）显示出显著的改善效果。通过实时预测模型与机器学习分类算法的结合，可以准确判断何时开启应急车道，并通过具体数据分析展示应急车道对交通流的显著改善效果。这些量化指标为交通管理者提供了科学依据，

能够更有效地缓解拥堵并提升道路通行效率。

七、 问题四

7.1 问题分析

为提升第三个点到第四个点之间路段应急车道临时启用决策的科学性，同时控制成本，优化视频监控点的布置应从以下几个方面考虑。为了有效监控 3 公里高速路段的交通流状态，需要合理安排监控点的位置，以覆盖整个路段并获取实时数据。考虑车流量的变化，覆盖关键路段，监控点间距优化，考虑覆盖与重点监控相结合。由于问题涉及多个目标（如成本、监控效果）以及复杂的约束条件（如预算、道路特征），使用遗传算法来优化监控点的布置。

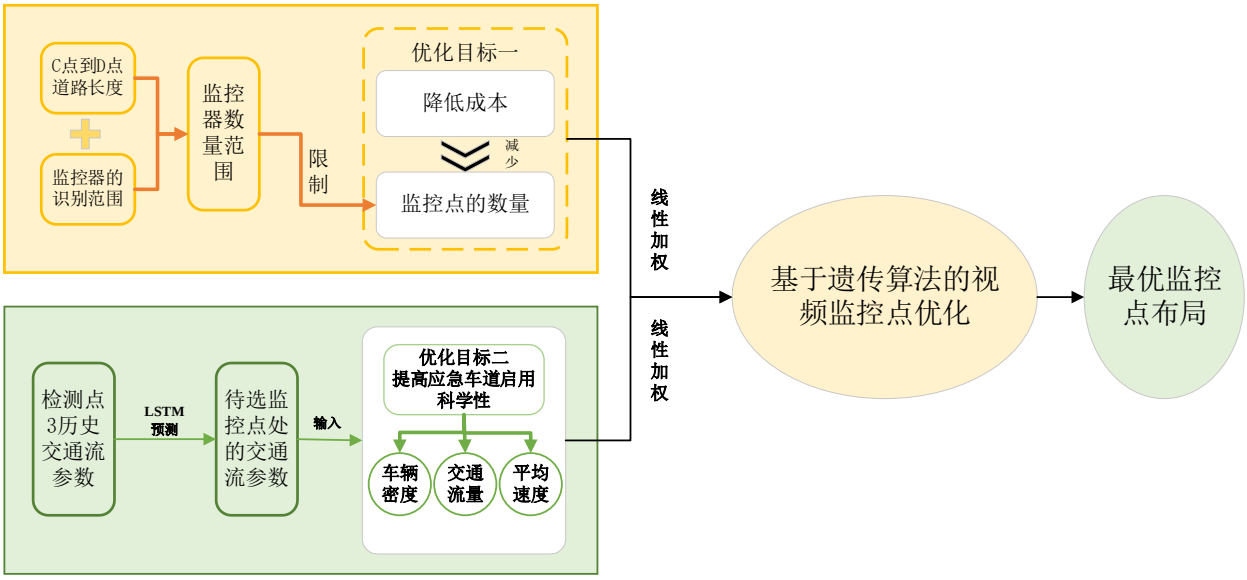

图 7-1　问题四技术路线图

7.2 目标优化模型建立

目标是在 3 公里的高速公路路段上，布置监控点，以实现实时应急车道启用的最佳决策。在有限的预算下，需要在离散的 15 个可能的监控点位置上（每隔 200 米），选择合适的监控点。布置的监控点需要收集车流量、车速、密度等数据，保证每 12 秒进行一次交通状态预测。

基于前文是以 2min 为采样点进行采集，在本小节，对第三个检测点的视频需要再次以 12s 为采样点进行采集。

设定以下决策变量：

- 位置选择变量 x_i：是否在第 i 个位置布置监控点。$x_i=1$ 表示布置，$x_i=0$ 表示不布置，i=1,2,3...,15。
- Q_C：流量（veh/h）
- K_C：密度（veh/km）
- V_C：平均速度（km/h）

目标函数：

$$G = b_1 \cdot \sum (U_i \cdot X_i) - b_2 \cdot \sum (P_i \cdot X_i) \tag{7-1}$$

- G：总目标函数。
- b_1 b_2：权重系数，用于平衡监控效果和成本。
- U_i：在位置 i 放置监控点的效果评分。
- P_i：在位置 i 放置监控点的成本。

监控点效果评分：

$$U_i = c_1 \cdot f(Q_i) + c_2 \cdot g(K_i) + c_3 \cdot h(V_i) \tag{7-2}$$

- c_1, c_2, c_3：权重系数，满足 $c_1 + c_2 + c_3 = 1$

- $f(Q_i)$ $g(K_i)$ $h(V_i)$ 分别是对车流量、密度、速度进行归一化处理。

预算约束：

$$\sum (P_i \cdot X_i) \le M \tag{7-3}$$

- M：表示总预算限制

监控点间距约束：

$$|i - j| \cdot (X_i + X_j) \ge d_{\min} \tag{7-4}$$

该约束确保任意两个监控点之间的最小间距不低于 $d_{\min}$。

监控点数量约束：

$$N \le \sum X_i \le O \tag{7-5}$$

- N 和 O：分别表示最小和最大监控点数量。

监控点覆盖要求：

$$\sum_{i=i}^{i+m-1} X_i \ge 1, \quad \forall i \in \{1, 2, \ldots, 16 - m\} \tag{7-6}$$

此约束确保在任意 m 个连续位置中至少有一个监控点进行覆盖。

遗传算法概述:

遗传算法通过模拟生物进化的过程，利用选择、交叉、变异等操作，逐步优化解。适

用于多目标、多约束的问题。

算法步骤：

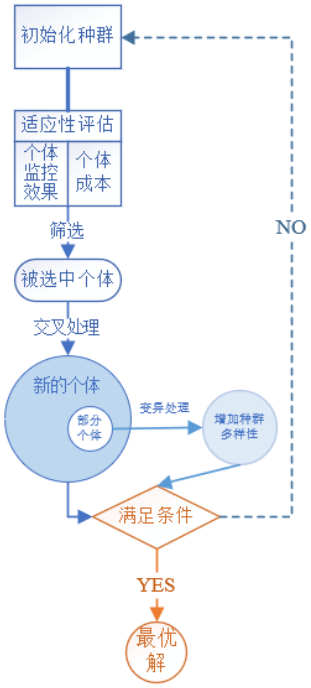

图 7-2　遗传算法的步骤

初始化种群：随机生成若干个体（解）。每个个体是一个由 x_i 变量构成的数组，代表某个方案下的监控点布置。

适应度函数：根据每个个体的监控效果和成本计算其适应度。

适应度函数基于目标函数 G，用来评估每个个体的优劣。适应度函数不仅考虑监控效果，还要考虑是否满足预算、间距等约束条件。假如某个个体违反了约束条件，则可以对其适应度施加惩罚项。

$$\text{Fitness}(X) = G(X) - \text{Penalty}(X) \tag{7-7}$$

其中，$\text{Fitness}(X)$ 是违反约束时的惩罚值，例如超出预算、违反间距限制或覆盖要求时罚分。

选择操作：使用轮盘赌选择或锦标赛选择来挑选适应度较高的个体参与下一代繁殖。通过这种方式，可以保留表现较好的个体，同时淘汰适应度较低的个体。

交叉操作:从两个父代个体中进行交叉操作，生成新的子代个体。可以使用单点交叉或多点交叉方法：

单点交叉：在两个父代个体中选择一个交叉点，交换其后续的基因片段，生成两个新个体。

多点交叉：选择多个交叉点，交换父代的片段，生成新的个体。

变异操作:在生成的子代中，随机选择某些基因进行变异。例如，将某个位置上的 0 变为 1，或将 1 变为 0。变异操作可以提高种群的多样性，防止算法陷入局部最优。

进化与收敛：通过选择、交叉和变异等操作，生成新一代种群。重复上述步骤多次，直到满足某个终止条件，例如达到最大代数或适应度值不再显著提升。

7.3 模型求解

在本次视频监控点布置优化问题中，采用了遗传算法来求解。通过遗传算法的不断迭代，选择、交叉、变异等操作，逐步优化监控点的布置，以最大化监控效果并最小化布置成本，同时满足预算、覆盖率和监控点间距的约束。模型迭代和求解：

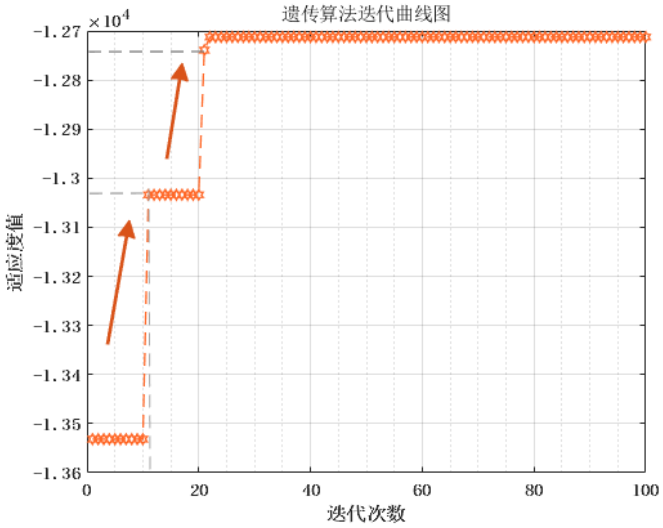

图 7-3　遗传算法迭代收敛曲线

从图中可以看出，遗传算法在大约 20 次迭代后基本收敛，适应度值不再有显著的提升。这意味着在 20 次迭代左右时，遗传算法找到了接近最优的监控点布置方案，在随后的迭代中，适应度值保持稳定，没有显著变化，表明算法已经接近全局最优解。。

关于视频监控点布置的方案：

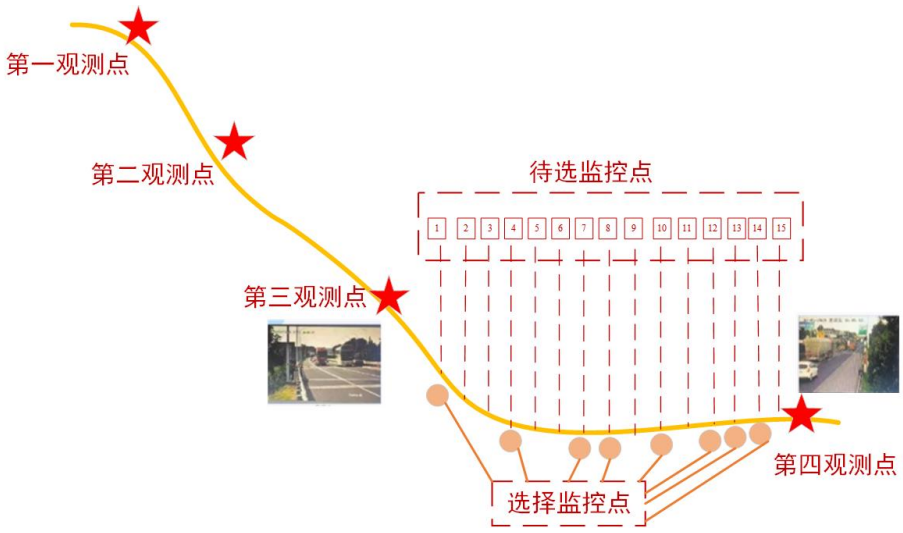

图 7-4　遗传算法优化结果

在靠近第四观测点的路段，图中标示 15 个待选监控点。这些监控点之间的距离较短，每个点之间相隔 200 米（根据前面描述的最小间距约束条件），这些待选监控点位置是遗传算法优化所要决定的目标。

圆形标记的监控点应该是经过遗传算法优化后的结果，即在满足预算、覆盖率、间距

约束的情况下，选择了这些监控点来优化监控效果和成本。

7.4 小结

为了实现最优布置，本文采用了遗传算法。遗传算法是一种基于自然选择的启发式优化算法，能够在复杂的多目标、多约束条件下进行全局搜索，并逐步逼近最优解。本次优化的目标是：

最大化监控点的布置效果评分，提升对交通流量、车速和密度的实时监控能力；

最小化总布置成本，确保方案在预算范围内；

满足监控点间的最小间距要求，避免监控资源浪费；

满足路段覆盖要求，确保所有重要路段都能被有效监控。

通过迭代收敛曲线分析，遗传算法在约 20 次迭代后基本收敛，找到了接近全局最优的监控点布置方案。在满足预算、间距和覆盖要求的前提下，算法选择了若干监控点，并通过最大化布置效果，实现了对关键路段的科学监控。

八、 模型推广

8.1 背景与现状分析

交通拥堵是全球范围内的大城市普遍面临的问题，尤其在节假日、高峰时段，交通流量激增导致的交通瓶颈尤为明显。传统的交通管理手段已经难以有效应对突发性和短时性的交通拥堵问题。

应急车道的临时启用，作为一种灵活应对拥堵的策略，已经在部分国家和地区得到了验证，并展现出一定的效果。然而，大多数现有的应急车道管理模式仍然依赖于固定时段的静态管理，未能充分结合实时交通数据进行动态优化。

8.2 模型的推广必要性

通过基于规则的机器学习模型，可以对实时交通数据进行动态分析和决策，从而实现应急车道的智能化启用，提升道路通行能力，减少拥堵发生的频率和时长。

该模型不仅适用于当前研究中所涉及的高速公路，还具有推广到其他类型道路和交通场景的潜力，如城市快速路、桥梁隧道等交通关键节点。

8.3 模型推广的可行性

技术可行性：

本文所构建的应急车道启用模型采用了多种先进的算法，包括 SVM、决策树、随机森林、AdaBoost、BP 神经网络和 Stacking 等。这些算法在大数据处理和实时决策方面表现出色，尤其是在处理复杂的交通流量和多维数据时，能够提供准确的预测和高效的决策支持。

模型可以与现有的交通监控系统无缝集成，依托摄像头、传感器等设备实时获取交通参数，借助大数据平台进行计算和存储，确保模型在实际应用中的实时性和可靠性。

应用场景广泛性：

该模型不仅适用于当前研究的具体场景（节假日高速公路交通管理），还可以推广到以下场景：

城市交通管理：城市快速路、主干道在高峰期面临的拥堵问题。

大型活动和赛事交通组织：在大型活动、体育赛事期间，短时交通流量激增，模型可用于指导交通疏导和应急车道启用。

应急救援和灾害管理：在突发自然灾害、事故情况下，模型可用于快速决策，保障救援通道的畅通。

8.4 社会效益

通过模型的推广应用，可以有效提升道路的通行能力，减少因拥堵带来的社会成本，包括时间损失、能源消耗和环境污染等。

同时，该模型的应用还能减少因交通拥堵引发的次生事故，提升道路安全性。

通过上述分析，本文所提出的基于规则的机器学习应急车道启用模型不仅具有较高的技术先进性和应用价值，还在社会效益方面表现突出。结合技术和管理措施，该模型有望在更大范围内推广应用，助力智慧交通的发展。

参考文献

[1] 熊亭，戚湧，张伟斌，等. 基于时空相关性的短时交通流预测模型[J]. 计算机工程与设计，2019，40（02）：501-507.

[2] 李瑞敏，张威威，刘志勇等. 组合段路段的长路段旅行时间短时预测[J]. 公路工程，2018，43（03）：1-5.

[3] 赵鹏，李璐. 基于 ARIMA 模型的城市轨道交通进站量预测研究[J]. 重庆交通大学学报（自然科学版），2020，39（01）：40-44.

[4] Ahmed M S,Cook A R. Analysis of freeway traffic time-series data by using Box-Jenkins techniques[J]. Transportation Research Record1979（722）.

[5] Johansson U,Bostrom HLöfström T, et al. Regression conformal prediction with random forests[J]. Machine learning, 2014, 97：155-176.

[6] 夏壬焕. 基于时空图神经网络的交通流预测研究[D]. 杭州：杭州电子科技大学,2023.

[7] 韩旭. 基于 LSTM 和 GCN 的交通流量预测研究[D]. 杭州：浙江科技学院,2023.

[8] 杨荣新. 基于机器学习的高速公路服务区交通量预测方法研究[D]. 西安：长安大学,2021.

[9] 汤立涛，莫杨辉. 基于 BP 神经网络与多元回归的公路客运量预测研究[J]. 交通科技,2017(5)：1-8.

[10] Siroos S, Milad G, Sisson S A, et al. Ensemble of ARIMA: combining parametric and bootstrapping technique for traffic flow prediction[J]. Transportmetrica A: Transport Science,2020,16(3)：15-26：.

[11] 王宏杰,林良明,徐大淦,等. 基于改进BP网交通流动态时序预测算法的研究[J]. 交通与计算机,2001(3)：1-4.

[12] 丁栋,朱云龙,库涛,等. 基于影响模型的短时交通流预测方法[J]. 计算机工程,2012,38(10)：1-4.

[13] 余涛. 基于 SVM 和 BP 神经网络的短时交通流预测与实现[D]. 南京:南京邮电大学,2018.

附录

问题一代码

```
import cv2  # 导入 OpenCV 库，用于处理视频和图像
import torch  # 导入 PyTorch 库，用于加载和使用深度学习模型
import numpy as np  # 导入 NumPy 库，用于数值计算
import pandas as pd  # 导入 Pandas 库，用于数据处理和保存

# 加载 YOLOv5 模型
model = torch.hub.load('ultralytics/yolov5', 'yolov5s')  # 使用 PyTorch 的 hub 功能加载 YOLOv5 模型，'yolov5s'表示加载的是 YOLOv5 的小型版本
def get_video_info(video_path):  # 定义一个函数，用于获取视频的基本信息
    cap = cv2.VideoCapture(video_path)  # 打开视频文件
    if not cap.isOpened():  # 检查视频是否成功打开
        print("无法打开视频文件")
        return None, None, None  # 如果无法打开视频，返回 None

    # 获取视频总帧数
    total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))  # 获取视频的总帧数
    # 获取视频帧率
    fps = cap.get(cv2.CAP_PROP_FPS)  # 获取视频的帧率（每秒帧数）
    # 获取视频时长（秒）
    duration = total_frames / fps  # 计算视频的时长（秒）
    print(f"视频总帧数: {total_frames}")  # 打印视频总帧数
    print(f"视频帧率: {fps}")  # 打印视频帧率
    print(f"视频时长（秒）: {duration}")  # 打印视频时长
    cap.release()  # 释放视频文件
    return total_frames, fps, duration  # 返回视频的总帧数、帧率和时长

def process_video(video_path, output_excel="traffic_data.xlsx", vehicle_length=4.0, safe_gap=2.0, frame_skip=10):  # 定义一个函数，用于处理视频并计算交通数据

    # 获取视频信息
    total_frames, fps, duration = get_video_info(video_path)  # 调用 get_video_info 函数获取视频信息
```

```
        if total_frames is None or fps is None or duration is None:  # 检查视频信息是否获取成功
            print("视频信息获取失败")
            return  # 如果获取失败，返回

        cap = cv2.VideoCapture(video_path)  # 重新打开视频文件
        if not cap.isOpened():  # 检查视频是否成功打开
            print("无法打开视频文件")
            return  # 如果无法打开视频，返回

        frame_count = 0  # 初始化帧计数器
        vehicle_counts = []  # 初始化车辆数量列表

        while True:  # 开始循环读取视频帧
            ret, frame = cap.read()  # 读取一帧视频
            if not ret:  # 检查是否成功读取帧
                break  # 如果读取失败，跳出循环
            # 仅处理每 frame_skip 帧中的一帧

            if frame_count % frame_skip != 0:  # 检查当前帧是否需要跳过
                frame_count += 1  # 增加帧计数器
                continue  # 跳过当前帧

            frame_count += 1  # 增加帧计数器

            # 使用 YOLO 模型进行车辆检测
            results = model(frame)  # 使用 YOLOv5 模型对当前帧进行车辆检测
            # 过滤检测结果，只保留车辆相关类别
            vehicles = results.pandas().xyxy[0][results.pandas().xyxy[0]['name'].isin(['car', 'truck', 'bus', 'motorbike'])]  # 过滤检测结果，只保留车辆类别（汽车、卡车、公交车、摩托车）
            # 统计当前帧的车辆数量

            vehicle_count = len(vehicles)  # 计算当前帧的车辆数量
            vehicle_counts.append(vehicle_count)  # 将车辆数量添加到列表中

            # 清理内存
```

```
            vehicles = None  # 释放 vehicles 变量占用的内存
            results = None  # 释放 results 变量占用的内存

        cap.release()  # 释放视频文件

        # 计算交通密度
        avg_vehicle_count = np.mean(vehicle_counts)  # 计算平均车辆数量
        road_length_estimate = avg_vehicle_count * (vehicle_length + safe_gap)  # 估算道路长度
        density = avg_vehicle_count / road_length_estimate  # 计算交通密度
        # 计算交通流量
        duration_per_frame = frame_skip / fps  # 每个采样的实际时间
        flow_rate = avg_vehicle_count / duration_per_frame  # 计算交通流量

        print(f"平均车辆数量: {avg_vehicle_count}")  # 打印平均车辆数量
        print(f"估算道路长度: {road_length_estimate:.2f} 米")  # 打印估算道路长度
        print(f"交通密度: {density:.4f} 辆/米")  # 打印交通密度
        print(f"交通流量: {flow_rate:.2f} 辆/秒")  # 打印交通流量
        # 保存结果到 Excel

        data = {

            "车辆数量": [avg_vehicle_count],  # 车辆数量
            "估算道路长度 (米)": [road_length_estimate],  # 估算道路长度
            "交通密度 (辆/米)": [density],  # 交通密度
            "交通流量 (辆/秒)": [flow_rate]  # 交通流量

        }
        df = pd.DataFrame(data)  # 将数据转换为 DataFrame
        df.to_excel(output_excel, index=False)  # 将数据保存到 Excel 文件
        print(f"数据已保存到 {output_excel}")  # 打印保存路径

    # 示例：使用视频文件进行处理并保存结果到 Excel

    video_path = r"D:\Desktop\JS_jiemi\高速公路交通流数据\32.31.250.103\20240501_20240501125647_20240501140806_125649.mp4"  # 替换为你的视频路径
```

output_excel = r"D:\Desktop\JS_jiemi\高速公路交通流数据\32.31.250.103/traffic_data.xlsx" # 替换为你的输出 Excel 路径

process_video(video_path, output_excel) # 调用 process_video 函数处理视频并保存结果

```
import cv2
import torch
from sort import Sort  # 导入 SORT 跟踪算法
import numpy as np

# 加载 YOLOv5 模型
model = torch.hub.load('ultralytics/yolov5', 'yolov5s', pretrained=True)

# SORT 跟踪器实例化
tracker = Sort()

# 用于存储每辆车的速度
vehicle_speeds = {}
# 用于存储每辆车的历史位置
vehicle_positions = {}

def process_video(video_path):
    cap = cv2.VideoCapture(video_path)
    fps = cap.get(cv2.CAP_PROP_FPS)  # 获取视频的帧率

    # 调整像素与实际距离的比例，假设视频中 1000 像素代表实际 75 米
    pixel_distance = 1500  # 假设视频中的一段路长是 1500 像素
    actual_distance = 100  # 实际路段长度为 100 米
    scale_factor = actual_distance / pixel_distance  # 每个像素代表多少米

    if not cap.isOpened():
        print("无法打开视频文件")
        return
    frame_count = 0

    while True:
        ret, frame = cap.read()
```

```
            if not ret:
                break
            frame_count += 1
            # 使用 YOLOv5 进行车辆检测
            results = model(frame)
            # 过滤检测结果，只保留车辆相关类别
            vehicles = results.pandas().xyxy[0][results.pandas().xyxy[0]['name'].isin(['car',
'truck', 'bus', 'motorbike'])]
            # 将 YOLO 检测结果转换为[左上角 X, 左上角 Y, 右下角 X, 右下角 Y, 置信
度]
            detections = vehicles[['xmin', 'ymin', 'xmax', 'ymax', 'confidence']].to_numpy()
            # 使用 SORT 进行车辆跟踪
            tracked_objects = tracker.update(detections)
            for obj in tracked_objects:
                x1, y1, x2, y2, track_id = obj[:5]  # 获取跟踪的边界框和跟踪 ID
                # 计算车辆的中心点位置
                center_x = (x1 + x2) / 2
                center_y = (y1 + y2) / 2
                # 如果车辆第一次出现，初始化它的位置信息
                if track_id not in vehicle_positions:
                    vehicle_positions[track_id] = {'previous_position': (center_x, center_y),
'frame_count': frame_count}
                else:
                    # 计算当前帧与上一帧之间的时间差
                    prev_x, prev_y = vehicle_positions[track_id]['previous_position']
                    previous_frame = vehicle_positions[track_id]['frame_count']
                    # 计算像素位移
                    distance_pixels = np.sqrt((center_x - prev_x) ** 2 + (center_y - prev_y)
** 2)
                    # 将像素位移转换为实际位移（米）
                    real_distance = distance_pixels * scale_factor
                    # 计算经过的时间（秒）
                    time_elapsed = (frame_count - previous_frame) / fps
                    if time_elapsed > 0:
                        # 计算速度（米/秒），并存储到 vehicle_speeds 字典中
                        speed_mps = real_distance / time_elapsed
                        speed_kmh = speed_mps * 3.6  # 转换为公里/小时
```

```
                    if track_id not in vehicle_speeds:
                        vehicle_speeds[track_id] = []

                    vehicle_speeds[track_id].append(speed_kmh)

                # 更新车辆位置和帧数
                vehicle_positions[track_id]['previous_position'] = (center_x, center_y)
                vehicle_positions[track_id]['frame_count'] = frame_count

            # 可视化跟踪框
            cv2.rectangle(frame, (int(x1), int(y1)), (int(x2), int(y2)), (0, 255, 0), 2)
            cv2.putText(frame, f'ID: {int(track_id)}', (int(x1), int(y1) - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.75,
                        (0, 255, 0), 2)

        # 显示处理后的帧
        cv2.imshow("Tracking", frame)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()

    # 计算所有车辆的平均速度，剔除速度超过 75 km/h 的车辆
    calculate_average_speed(vehicle_speeds)

def calculate_average_speed(vehicle_speeds):
    total_speed = 0
    total_cars = 0

    # 计算每辆车的平均速度并剔除速度超过 75 km/h 的车辆
    for track_id, speeds in vehicle_speeds.items():
        if len(speeds) > 0:
            # 剔除速度大于 75 km/h 的数据
            filtered_speeds = [speed for speed in speeds if speed <= 75]
            if len(filtered_speeds) > 0:
```

```
                    avg_speed = np.mean(filtered_speeds)
                    total_speed += avg_speed
                    total_cars += 1
                    print(f'Vehicle ID: {int(track_id)}, Average Speed: {avg_speed:.2f} km/h')

    # 计算视频中所有车辆的平均速度
    if total_cars > 0:
        overall_average_speed = total_speed / total_cars
        print(f'Overall Average Speed of All Vehicles: {overall_average_speed:.2f} km/h')
    else:
        print("No vehicles tracked below 75 km/h.")


# 视频路径
video_path = r"D:\Desktop\JS_jiemi\高速公路交通流数据\32.31.250.103\20240501_20240501125647_20240501140806_125649.mp4"
process_video(video_path)


import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error
import itertools
import warnings
import matplotlib.font_manager as fm

# 关闭警告
warnings.filterwarnings("ignore")

# 设置中文字体支持
plt.rcParams['font.sans-serif'] = ['SimHei']  # 设置字体为黑体，解决中文显示问题
plt.rcParams['axes.unicode_minus'] = False  # 解决坐标轴负号显示问题
```

```
    # TPI 数据
    TPI_data = np.array([
        3.402001139, 3.047938016, 3.100207732, 3.388036556, 3.294612631, 2.942086967,
3.665145391, 3.620222999,
        3.529716666, 3.351145552, 3.367576895, 2.909688516, 3.830687934, 3.125863087,
3.406666967, 3.382291516,
        3.296774458, 3.418176377, 3.621982356, 3.239540894, 3.882916133, 3.086680856,
3.302424755, 3.981660606,
        3.361716495, 2.500200535, 3.303352199, 3.33763333, 3.548847015, 3.328671527,
3.244503097, 3.126723444,
        2.719523929, 3.095535844, 3.082043424, 3.50279618, 2.635263361, 3.220496385,
3.539243198, 3.198456586,
        3.253053908, 3.438614484, 3.852343179, 2.850283809, 2.845233304, 2.67545442,
2.9315247, 3.199626616,
        3.36652888, 3.534654665, 2.827191565, 2.880779775, 3.761740192, 3.695583128,
2.593329472, 3.671607495,
        3.389842405, 3.460424482, 2.730759288, 3.460958021, 3.406413661, 2.565159446,
2.690557944, 2.892452332,
        3.038353192, 2.753146198, 3.294384201, 3.966404947
    ])

    # 数据转换为 pandas series
    tpi_series = pd.Series(TPI_data)

    # 迭代搜索 ARIMA 的 p 和 q
    p = q = range(0, 5)  # 设置范围
    pq_combinations = list(itertools.product(p, q))
    aic_results = {}

    for param in pq_combinations:
        try:
            model = ARIMA(tpi_series, order=(param[0], 1, param[1]))
            model_fit = model.fit()
            aic_results[param] = model_fit.aic
        except:
            continue

    # 选择最优的 p, q 参数
```

```
best_params = min(aic_results, key=aic_results.get)
print(f"最优的 p, q 参数: {best_params}")

# 绘制 AIC 热力图（蓝色调）
aic_matrix = np.zeros((5, 5))
for param, aic in aic_results.items():
    aic_matrix[param[0], param[1]] = aic

plt.figure(figsize=(10, 8))
sns.heatmap(aic_matrix, annot=True, fmt='.2f', cmap='Blues', xticklabels=p, yticklabels=q)
plt.title('ARIMA 模型的 AIC 热力图')
plt.xlabel('p 参数')
plt.ylabel('q 参数')
plt.savefig('arima_aic_heatmap.png')
plt.show()

# 基于最优 p, q 参数建立 ARIMA 模型
best_model = ARIMA(tpi_series, order=(best_params[0], 1, best_params[1]))
best_model_fit = best_model.fit()

# 预测未来 30 分钟的 TPI 值
forecast = best_model_fit.forecast(steps=15)
print("未来 30 分钟的 TPI 预测值：")
print(forecast)

# 预测结果标签化
def classify_tpi(value):
    if value < 1:
        return '畅通'
    elif 1 <= value < 2:
        return '基本畅通'
    elif 2 <= value < 3:
        return '缓行'
    elif 3 <= value < 4:
        return '轻度拥堵'
    elif 4 <= value < 5:
        return '拥堵'
```

```
        else:
            return '严重拥堵'

    # 给预测的 TPI 值打标签
    forecast_labels = [classify_tpi(val) for val in forecast]
    print("未来 30 分钟的 TPI 预测标签：")
    print(forecast_labels)

    # 可视化预测结果
    plt.figure(figsize=(10, 6))
    plt.plot(tpi_series, label='历史 TPI 值')
    plt.plot(range(len(tpi_series), len(tpi_series) + 15), forecast, label='预测 TPI 值', color='red')
    plt.xlabel('时间点')
    plt.ylabel('TPI 值')
    plt.title('未来 30 分钟的 TPI 预测')
    plt.legend()
    plt.savefig('tpi_forecast.png')
    plt.show()

    # 打印最后 15 行预测的 TPI 值和对应的标签
    predicted_data = pd.DataFrame({
        'TPI 预测值': forecast,
        '预测标签': forecast_labels
    })
    print(predicted_data.head(15))

    # 将结果导出为 Excel
    predicted_data.to_excel('TPI 预测结果.xlsx', index=False)
    import numpy as np
    import pandas as pd
    from sklearn.metrics import accuracy_score, recall_score, f1_score, mean_squared_error
    from keras.models import Sequential
    from keras.layers import LSTM, Dense
    from sklearn.preprocessing import MinMaxScaler
    from pyswarm import pso  # 导入 PSO 优化算法
    import matplotlib.pyplot as plt
```

```
# 设置中文字体
plt.rcParams['font.sans-serif'] = ['SimHei']  # 解决中文显示问题
plt.rcParams['axes.unicode_minus'] = False  # 解决负号显示问题

# 加载数据集
data = pd.read_excel('第三段时间序列分析数据.xlsx')

# 将数据集中相关列提取为数组
speed_data = data['平均速度 (千米/小时)'].values.reshape(-1, 1)
density_data = data['车辆密度 (辆/公里)'].values.reshape(-1, 1)
true_labels = data['拥堵等级'].values

# 定义类别映射
label_mapping = {'基本畅通': 0, '缓行': 1, '轻度拥堵': 2, '拥堵': 3, '严重拥堵': 4}
true_labels = np.array([label_mapping[label] for label in true_labels])

# 数据归一化
scaler_speed = MinMaxScaler()
scaler_density = MinMaxScaler()
scaled_speed = scaler_speed.fit_transform(speed_data)
scaled_density = scaler_density.fit_transform(density_data)

# 准备 LSTM 训练数据，使用过去 30 个数据点预测下一个数据点
def create_dataset(dataset, time_step=30):
    X, y = [], []
    for i in range(len(dataset) - time_step - 1):
        a = dataset[i:(i + time_step), 0]
        X.append(a)
        y.append(dataset[i + time_step, 0])
    return np.array(X), np.array(y)

# 创建车辆密度和平均速度的训练集
time_step = 30
X_speed, y_speed = create_dataset(scaled_speed, time_step)
X_density, y_density = create_dataset(scaled_density, time_step)
```

```
    # 重塑输入以适应 LSTM 的输入格式 [samples, time_steps, features]
    X_speed = X_speed.reshape(X_speed.shape[0], X_speed.shape[1], 1)
    X_density = X_density.reshape(X_density.shape[0], X_density.shape[1], 1)

    # 构建 LSTM 模型
    def build_lstm_model():
        model = Sequential()
        model.add(LSTM(50, return_sequences=True, input_shape=(time_step, 1)))
        model.add(LSTM(50))
        model.add(Dense(1))  # 预测一个值
        model.compile(loss='mean_squared_error', optimizer='adam')
        return model

    # 创建并训练车辆密度和速度的 LSTM 模型
    model_speed = build_lstm_model()
    model_density = build_lstm_model()

    # 训练模型
    model_speed.fit(X_speed, y_speed, epochs=20, batch_size=32, verbose=1)
    model_density.fit(X_density, y_density, epochs=20, batch_size=32, verbose=1)

    # 预测车辆密度和速度
    pred_speed = model_speed.predict(X_speed)
    pred_density = model_density.predict(X_density)

    # 逆归一化，恢复真实的车辆密度和速度值
    pred_speed = scaler_speed.inverse_transform(pred_speed)
    pred_density = scaler_density.inverse_transform(pred_density)

    # 定义分类函数，基于车辆密度和速度的阈值分类
    def classify(speed, density, thresholds):
        speed_th1, speed_th2, speed_th3, speed_th4, density_th1, density_th2, density_th3,
density_th4 = thresholds
        if speed > speed_th1 and density < density_th1:
            return 0  # 基本畅通
        elif speed_th2 < speed <= speed_th1 and density_th2 < density <= density_th1:
            return 1  # 缓行
```

```
    elif speed_th3 < speed <= speed_th2 and density_th3 < density <= density_th2:
        return 2  # 轻度拥堵
    elif speed_th4 < speed <= speed_th3 and density_th4 < density <= density_th3:
        return 3  # 拥堵
    else:
        return 4  # 严重拥堵

# 定义适应度函数
def fitness(thresholds):
    pred_labels = []
    for speed, density in zip(pred_speed, pred_density):
        pred_labels.append(classify(speed, density, thresholds))
    accuracy = accuracy_score(true_labels[:len(pred_labels)], pred_labels)
    return -accuracy  # PSO 算法是最小化目标，因此使用负的准确率

# 设置阈值的上下界
lb = [30, 30, 30, 30, 80, 80, 80, 80]  # 下界
ub = [100, 100, 100, 100, 130, 130, 130, 130]  # 上界

# 使用 PSO 寻找最优阈值
best_thresholds, best_accuracy = pso(fitness, lb, ub, swarmsize=50, maxiter=100)

# 使用最优阈值对 LSTM 预测的数据进行分类
pred_labels = []
for speed, density in zip(pred_speed, pred_density):
    pred_labels.append(classify(speed, density, best_thresholds))

# 计算分类准确率、召回率、F1 分数
accuracy = accuracy_score(true_labels[:len(pred_labels)], pred_labels)
recall = recall_score(true_labels[:len(pred_labels)], pred_labels, average='macro')
f1 = f1_score(true_labels[:len(pred_labels)], pred_labels, average='macro')

print(f"分类准确率: {accuracy}")
print(f"召回率: {recall}")
print(f"F1 分数: {f1}")

# LSTM 预测误差
```

```
mse_speed = mean_squared_error(speed_data[:len(pred_speed)], pred_speed)
mse_density = mean_squared_error(density_data[:len(pred_density)], pred_density)
print(f"平均速度预测误差（MSE）: {mse_speed}")
print(f"车辆密度预测误差（MSE）: {mse_density}")

# 可视化真实标签 vs 预测标签
plt.figure(figsize=(10, 6))
plt.plot(true_labels[:len(pred_labels)], label='真实标签', linestyle='-', marker='o')
plt.plot(pred_labels, label='预测标签', linestyle='--', marker='x')
plt.xlabel('样本索引')
plt.ylabel('拥堵等级')
plt.title('真实标签 vs 预测标签')
plt.legend()
plt.savefig('lstm_predicted_vs_true.png')
plt.show()

# 可视化 LSTM 预测的平均速度和车辆密度
plt.figure(figsize=(10, 6))
plt.plot(speed_data[:len(pred_speed)], label='真实平均速度', linestyle='-', marker='o')
plt.plot(pred_speed, label='预测平均速度', linestyle='--', marker='x')
plt.xlabel('样本索引')
plt.ylabel('平均速度 (千米/小时)')
plt.title('真实平均速度 vs 预测平均速度')
plt.legend()
plt.savefig('lstm_speed_predictions.png')
plt.show()

plt.figure(figsize=(10, 6))
plt.plot(density_data[:len(pred_density)], label='真实车辆密度', linestyle='-', marker='o')
plt.plot(pred_density, label='预测车辆密度', linestyle='--', marker='x')
plt.xlabel('样本索引')
plt.ylabel('车辆密度 (辆/公里)')
plt.title('真实车辆密度 vs 预测车辆密度')
plt.legend()
plt.savefig('lstm_density_predictions.png')
plt.show()
```

```
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score
from pyswarm import pso
import matplotlib.pyplot as plt
import matplotlib.font_manager as fm

# 设置中文字体
plt.rcParams['font.sans-serif'] = ['SimHei']  # 解决中文显示问题
plt.rcParams['axes.unicode_minus'] = False  # 解决负号显示问题

# 加载数据集
data = pd.read_excel('第三段时间序列分析数据.xlsx')

# 将数据集中相关列提取为数组
speed_data = data['平均速度 (千米/小时)'].values
density_data = data['车辆密度 (辆/公里)'].values
true_labels = data['拥堵等级'].values

# 定义类别映射，增加五个类别
label_mapping = {'基本畅通': 0, '缓行': 1, '轻度拥堵': 2, '拥堵': 3, '严重拥堵': 4}

# 将标签映射为数值
true_labels = np.array([label_mapping[label] for label in true_labels])

# 定义分类函数，基于五个类别的阈值
def classify(speed, density, thresholds):
    speed_th1, speed_th2, speed_th3, speed_th4, density_th1, density_th2, density_th3, density_th4 = thresholds
    if speed > speed_th1 and density < density_th1:
        return 0  # 基本畅通
    elif speed_th2 < speed <= speed_th1 and density_th2 < density <= density_th1:
        return 1  # 缓行
    elif speed_th3 < speed <= speed_th2 and density_th3 < density <= density_th2:
        return 2  # 轻度拥堵
    elif speed_th4 < speed <= speed_th3 and density_th4 < density <= density_th3:
        return 3  # 拥堵
```

```
        else:
            return 4  # 严重拥堵

    # 定义适应度函数
    def fitness(thresholds):
        predictions = []
        for speed, density in zip(speed_data, density_data):
            predictions.append(classify(speed, density, thresholds))
        accuracy = accuracy_score(true_labels, predictions)
        return -accuracy  # 负的准确率，因为 pso 是最小化问题

    # PSO 每次迭代时保存当前的准确率
    accuracy_history = []

    def fitness_with_history(thresholds):
        global accuracy_history
        accuracy = -fitness(thresholds)
        accuracy_history.append(accuracy)
        return -accuracy

    # 设置阈值的上下界 (包括四个速度和四个密度阈值)
    lb = [20, 30, 40, 50, 60, 70, 80, 90]  # 下界：四个速度和四个密度的下界
    ub = [100, 90, 80, 70, 140, 130, 120, 110]  # 上界：四个速度和四个密度的上界

    # 使用粒子群优化算法寻找最佳阈值，并返回所有迭代的准确率
    best_thresholds, best_accuracy = pso(fitness_with_history, lb, ub, swarmsize=30, maxiter=50)

    # 输出最优结果
    print(f"最优阈值: {best_thresholds}")
    print(f"最优准确率: {-best_accuracy}")

    # 使用最优阈值进行预测
    predictions = []
    for speed, density in zip(speed_data, density_data):
        predictions.append(classify(speed, density, best_thresholds))
```

```
# 添加预测结果到数据集
data['预测拥堵等级'] = predictions
data['预测拥堵等级'] = data['预测拥堵等级'].map({0: '基本畅通', 1: '缓行', 2: '轻度拥堵', 3: '拥堵', 4: '严重拥堵'})

# 保存结果到 Excel
data.to_excel('pso_predicted_results.xlsx', index=False)

# 可视化真实标签 vs 预测标签
plt.figure(figsize=(10, 6))
plt.plot(data.index, true_labels, label='真实标签', linestyle='-', marker='o')
plt.plot(data.index, predictions, label='预测标签', linestyle='--', marker='x')
plt.xlabel('样本索引')
plt.ylabel('拥堵等级')  # 使用数值来表示不同拥堵等级
plt.title('真实标签 vs 预测标签')
plt.yticks([0, 1, 2, 3, 4], ['基本畅通', '缓行', '轻度拥堵', '拥堵', '严重拥堵'])  # 设置 Y 轴标签
plt.legend()
plt.savefig('pso_predicted_vs_true.png')
plt.show()

# 绘制准确率随迭代次数的变化曲线
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(accuracy_history) + 1), accuracy_history, marker='o')
plt.xlabel('迭代次数')
plt.ylabel('准确率')
plt.title('粒子群优化迭代准确率变化')
plt.grid(True)
plt.savefig('pso_accuracy_iterations.png')
plt.show()
```

问题三代码

```
# 导入必要的库
import pandas as pd
import numpy as np
```

```
from sklearn.model_selection import train_test_split, cross_val_predict, StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, recall_score, f1_score, confusion_matrix, roc_curve, auc
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier, StackingClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import LogisticRegression
import matplotlib.pyplot as plt
import seaborn as sns

# 读取数据
file_path = '是否开启应急车道结果.xlsx'  # 请替换为你的文件路径
data = pd.read_excel(file_path)

# 数据预处理：将开启拥堵状态编码为 0 和 1
data['开启拥堵状态'] = data['开启拥堵状态'].astype(int)

# 选择自变量和因变量
X = data[['平均速度 (千米/小时)', '车辆密度 (辆/公里)', '交通流量（辆/小时）', '拥堵等级编号']]
y = data['开启拥堵状态']

# 拆分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# 标准化数据
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 初始化模型
models = {
    'SVM': SVC(probability=True, random_state=42),
    'Decision Tree': DecisionTreeClassifier(random_state=42),
```

```
        'Random Forest': RandomForestClassifier(random_state=42),
        'AdaBoost': AdaBoostClassifier(random_state=42),
        'BP Neural Network': MLPClassifier(random_state=42),
    }

    # 创建 Stacking 模型
    estimators = [
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(probability=True, random_state=42)),
        ('dt', DecisionTreeClassifier(random_state=42))
    ]
    stacking_model = StackingClassifier(estimators=estimators,
final_estimator=LogisticRegression())
    models['Stacking'] = stacking_model

    # 进行 5 折交叉验证
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

    # 保存交叉验证结果
    cv_results = []
    confusion_matrices_cv = []
    roc_curves_cv = {}

    for name, model in models.items():
        # Cross-validation predictions
        y_pred_cv = cross_val_predict(model, X, y, cv=cv, method='predict')
        y_prob_cv = cross_val_predict(model, X, y, cv=cv, method='predict_proba')[:, 1]

        # 计算评价指标
        accuracy_cv = accuracy_score(y, y_pred_cv)
        recall_cv = recall_score(y, y_pred_cv)
        f1_cv = f1_score(y, y_pred_cv)

        # 保存评价指标
        cv_results.append({
            'Model': name,
            'Accuracy': accuracy_cv,
```

```
            'Recall': recall_cv,
            'F1 Score': f1_cv
        })

        # 混淆矩阵
        confusion_matrices_cv.append(confusion_matrix(y, y_pred_cv))

        # ROC 曲线
        fpr_cv, tpr_cv, _ = roc_curve(y, y_prob_cv)
        roc_auc_cv = auc(fpr_cv, tpr_cv)
        roc_curves_cv[name] = (fpr_cv, tpr_cv, roc_auc_cv)

# 展示交叉验证的结果
cv_results_df = pd.DataFrame(cv_results)
print(cv_results_df)

# 画出混淆矩阵
fig, axes = plt.subplots(3, 2, figsize=(12, 12))
axes = axes.ravel()

for i, (name, matrix_cv) in enumerate(zip(models.keys(), confusion_matrices_cv)):
    sns.heatmap(matrix_cv, annot=True, fmt="d", ax=axes[i], cmap="Blues")
    axes[i].set_title(f"{name} CV Confusion Matrix")
    axes[i].set_xlabel("Predicted")
    axes[i].set_ylabel("Actual")

plt.tight_layout()
plt.show()

# 画出 ROC 曲线
plt.figure(figsize=(10, 8))
for name, (fpr_cv, tpr_cv, roc_auc_cv) in roc_curves_cv.items():
    plt.plot(fpr_cv, tpr_cv, label=f'{name} (AUC = {roc_auc_cv:.2f})')

plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
```

```
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Cross-Validation ROC Curve')
plt.legend(loc='lower right')
plt.show()

#1.根据阈值准则判断是否需要开启应急车道
import pandas as pd

# 创建数据框

df=pd.read_excel('总和数据.xlsx')
# 定义拥堵状态编号
congestion_levels = {
    '基本畅通': 1,
    '缓行': 2,
    '轻度拥堵': 3,
    '拥堵': 4,
    '严重拥堵': 5
}

# 将拥堵等级映射到编号
df['拥堵等级编号'] = df['拥堵等级'].map(congestion_levels)

# 定义阈值条件
flow_threshold = 5000
density_threshold = 140
speed_threshold = 30
level_threshold = 4



# 定义是否开启拥堵状态的函数
def check_congestion(row):
    conditions = [
        row['交通流量（辆/小时）'] > flow_threshold,  # 流量大于 5000
        row['车辆密度 (辆/公里)'] > density_threshold,  # 密度大于 140
        row['平均速度 (千米/小时)'] < speed_threshold,  # 速度小于 30
```

```
        row['拥堵等级编号'] >= level_threshold  # 拥堵状态编号大于等于 4
    ]

    # 至少满足三个条件
    if sum(conditions) >= 3:
        return True
    else:
        return False


# 应用函数，判断是否开启拥堵状态
df['开启拥堵状态'] = df.apply(check_congestion, axis=1)
df.to_excel("是否开启应急车道结果.xlsx")
# 显示结果
print(df[['平均速度 (千米/小时)', '车辆密度 (辆/公里)', '交通流量（辆/小时）', '拥堵等级', '开启拥堵状态']])
```

问题四代码

```
import numpy as np
import random
import matplotlib.pyplot as plt

# 参数设置
num_locations = 15  # 总共 15 个观测点
max_monitor_points = 15  # 允许的最大监控点数
min_monitor_points = 8  # 允许的最小监控点数
max_cost = 300000  # 最大预算
min_distance = 2  # 最小监控点间距（以 100 米为单位）

# 监控点参数数据（速度、密度、流量）
speed = np.array(
    [58.44, 54.03, 54.13, 57.56, 52.53, 54.47, 53.46, 59.38, 56.39, 60.04, 57.24, 53.24, 55.94, 76.27, 66.20])
density = np.array([82, 85, 85, 83, 91, 86, 87, 90, 82, 88, 83, 86, 85, 72, 75])
flow = np.array(
    [4791.93, 4592.53, 4600.68, 4777.51, 4780.45, 4684.51, 4650.82, 5344.48, 4624.35, 5283.10, 4750.69, 4578.81,
```

```
        4754.56, 5491.30, 4965.05])

# 成本数据（假设随机生成的成本）
# cost = np.random.randint(50, 150, size=num_locations)
cost = np.random.randint(10000, 30000, size=num_locations)

# 归一化函数，避免除以 0 的情况
def normalize(x):
    min_x = np.min(x)
    max_x = np.max(x)
    if max_x == min_x:
        return np.zeros_like(x)  # 如果所有值相同，返回全 0
    return (x - min_x) / (max_x - min_x)

# 适应度函数（计算效果评分和成本）
def fitness(individual):
    selected_indices = np.where(individual == 1)[0]
    if len(selected_indices) == 0:
        return -100  # 如果没有选中任何监控点，适应度为一个较低值
    # 计算效果评分
    f_speed = normalize(speed[selected_indices])
    f_density = normalize(density[selected_indices])
    f_flow = normalize(flow[selected_indices])
    E = 0.3 * f_speed + 0.3 * f_density + 0.4 * f_flow  # 权重系数分别为 0.3, 0.3, 0.4
    total_effect = np.sum(E)

    # 计算总成本
    total_cost = np.sum(cost[selected_indices])

    # 引入适应度基线，避免适应度为 0
    w1, w2 = 1, 0.1
    penalty = 0

    # 预算约束惩罚
    if total_cost > max_cost:
        penalty += (total_cost - max_cost) * 0.05  # 惩罚项（允许轻微超预算）
```

```
        # 检查监控点间距
        if not valid(individual):
            penalty += 50  # 违反监控点间距的惩罚

        return w1 * total_effect - w2 * total_cost - penalty  # 加上惩罚项

# 生成初始种群时确保监控点数量在允许范围内
def generate_population(pop_size):
    population = []
    for _ in range(pop_size):
        individual = np.zeros(num_locations)
        num_points = random.randint(min_monitor_points, max_monitor_points)
        selected_indices = random.sample(range(num_locations), num_points)
        individual[selected_indices] = 1
        population.append(individual)
    return np.array(population)

# 选择操作（轮盘赌）
def selection(population, fitness_values):
    fitness_values = np.nan_to_num(fitness_values, nan=0.1, posinf=0.1, neginf=0.1)
    if np.all(fitness_values <= 0):
        fitness_values = np.ones_like(fitness_values)  # 避免总权重为 0 的情况
    selected = random.choices(population, weights=fitness_values, k=len(population))
    return np.array(selected)

# 交叉操作
def crossover(parent1, parent2):
    crossover_point = np.random.randint(0, num_locations)
    child1 = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
    child2 = np.concatenate((parent2[:crossover_point], parent1[crossover_point:]))
    return repair(child1), repair(child2)

# 变异操作
def mutation(individual, mutation_rate=0.01):
    for i in range(len(individual)):
        if np.random.rand() < mutation_rate:
            individual[i] = 1 - individual[i]  # 翻转 0 和 1
```

```
        return repair(individual)

    # 修正个体，确保监控点数量在允许范围内
    def repair(individual):
        num_selected = np.sum(individual)
        if num_selected < min_monitor_points:
            additional_indices = random.sample(list(np.where(individual == 0)[0]), min_monitor_points - int(num_selected))
            individual[additional_indices] = 1
        elif num_selected > max_monitor_points:
            remove_indices = random.sample(list(np.where(individual == 1)[0]), int(num_selected) - max_monitor_points)
            individual[remove_indices] = 0
        return individual

    # 约束条件：保证监控点数量和间距
    def valid(individual):
        num_selected = np.sum(individual)
        if num_selected < min_monitor_points or num_selected > max_monitor_points:
            return False
        selected_indices = np.where(individual == 1)[0]
        for i in range(len(selected_indices) - 1):
            if selected_indices[i + 1] - selected_indices[i] < min_distance:
                return False
        return True

    # 遗传算法主程序
    def genetic_algorithm(pop_size=50, generations=100):
        population = generate_population(pop_size)
        best_individual = None
        best_fitness = -np.inf
        fitness_history = []

        for gen in range(generations):
            # 计算适应度值
            fitness_values = np.array([fitness(ind) for ind in population])
            best_gen_fitness = np.max(fitness_values)
```

```
            best_gen_individual = population[np.argmax(fitness_values)]

            # 更新最佳个体
            if best_gen_fitness > best_fitness:
                best_fitness = best_gen_fitness
                best_individual = best_gen_individual

            # 选择、交叉和变异
            selected_population = selection(population, fitness_values)
            new_population = []
            for i in range(0, len(selected_population), 2):
                parent1, parent2 = selected_population[i], selected_population[i + 1]
                child1, child2 = crossover(parent1, parent2)
                new_population.append(mutation(child1))
                new_population.append(mutation(child2))
            population = np.array(new_population)

            fitness_history.append(best_fitness)
            print(f"Generation {gen + 1}: Best fitness = {best_fitness}")

        # 绘制迭代曲线
        plt.figure(figsize=(10, 6))
        plt.plot(fitness_history, label="Best Fitness")
        plt.xlabel("Generation")
        plt.ylabel("Fitness")
        plt.title("Evolution of Fitness Over Generations")
        plt.legend()
        plt.grid(True)
        plt.savefig('fitness_evolution.png')
        plt.show()

        return best_individual, best_fitness, fitness_history

# 运行遗传算法
best_solution, best_solution_fitness, fitness_history = genetic_algorithm()
selected_monitor_points = np.where(best_solution == 1)[0]
```

```
# 输出最优解
print("最优监控点布局:", selected_monitor_points)
print("最优适应度值:", best_solution_fitness)

# 导出适应度结果
np.savetxt('fitness_history.csv', fitness_history, delimiter=',', header='Fitness')
```