**【优化类】创新型算法汇总**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **编号** | **赛题类型** | **组合创新** | **主要优势** | **常见案例** |
| 1 | 遗传算法（GA）+ 模拟退火（SA） | 结合 GA 的全局搜索能力与 SA 的局部寻优能力，避免过早收敛，提高解的精度 | 物流路径优化、车间调度、函数优化 | 适合需要高精度最优解且易出现早熟收敛的复杂优化问题 |
| 2 | 粒子群优化（PSO）+ 禁忌搜索（TS） | PSO 快速全局寻优，TS 通过禁忌表避免重复搜索，增强对离散问题的适配性 | 生产调度、电网负荷分配、网络路由优化 | 适合离散优化问题，且需要高效搜索并避免重复解的场景 |
| 3 | 蚁群算法（ACO）+ 贪婪算法（Greedy） | ACO 全局路径搜索，Greedy 快速生成初始解并局部优化，加速收敛 | 旅行商问题（TSP）、无人机巡检路径规划 | 适合路径规划类问题，尤其是需要快速收敛到较优解的场景 |
| 4 | 支持向量机（SVM）+ 遗传算法（GA） | GA 优化 SVM 的核函数参数和惩罚因子，提升 SVM 的分类 / 回归精度与泛化能力 | 化工反应参数优化、故障诊断中的阈值优化 | 适合依赖 SVM 进行分类或回归，且其性能高度依赖参数选择的问题 |
| 5 | 模糊逻辑（FL）+ 粒子群优化（PSO） | FL 处理不确定性信息（如模糊约束），PSO 高效寻优，适合含模糊变量的问题 | 交通信号配时、环境污染物减排优化 | 适合含有不确定性信息或模糊约束的优化问题 |
| 6 | 深度学习（DNN）+ 梯度下降（GD） | DNN 拟合复杂非线性关系，GD 优化网络参数，提高对高维数据的优化能力 | 能源负荷预测与调度、金融资产配置 | 适合高维数据下的复杂非线性关系拟合与参数优化问题 |
| 7 | 多目标遗传算法（MOGA）+ 层次分析法（AHP） | MOGA 生成多目标帕累托最优解，AHP 量化主观权重，实现多目标决策平衡 | 工程方案选型（成本 / 质量 / 工期）、供应链多目标优化 | 适合需要平衡多个目标，且需融入主观偏好进行决策的多目标优化问题 |
| 8 | 模拟退火（SA）+ 线性规划（LP） | SA 处理非线性 / 离散问题，LP 对线性子问题精确求解，兼顾全局与局部最优 | 救灾物资分配、混合整数规划问题 | 适合包含线性子问题和非线性 / 离散部分的混合优化问题 |
| 9 | 贝叶斯优化（BO）+ 随机森林（RF） | BO 高效搜索超参数空间，RF 构建代理模型减少真实函数评估成本，适合昂贵优化 | 材料配方优化（实验成本高）、工业设备参数调优 | 适合优化过程中真实函数评估成本高昂的昂贵优化问题 |
| 10 | 禁忌搜索（TS）+ 0-1 整数规划（IP） | TS 处理离散组合问题，IP 对整数约束精确求解，增强解的可行性 | 仓库 / 基站选址、投资组合选择 | 适合具有 0-1 整数约束的离散组合优化问题，强调解的可行性 |
| 11 | 人工蜂群算法（ABC）+ 差分进化（DE） | ABC 的全局勘探能力与 DE 的局部开发能力结合，提高收敛速度和解的多样性 | 函数优化、机器人路径规划、化工过程优化 | 适合需要平衡全局勘探与局部开发，追求收敛速度和解的多样性的优化问题 |
| 12 | 模糊神经网络（FNN）+ 粒子群优化（PSO） | FNN 处理模糊信息和非线性关系，PSO 优化网络权重和结构，提升预测与优化能力 | 污水处理参数优化、医疗资源调度（含不确定需求） | 适合含模糊信息和复杂非线性关系的预测与优化问题，尤其存在不确定需求时 |
| 13 | 多目标粒子群优化（MOPSO）+ 熵权法（EWM） | MOPSO 生成多目标解，EWM 客观赋权避免主观偏差，适合多指标平衡问题 | 湿地开发（生态与经济平衡）、物流多目标（成本 / 时效 / 碳排放） | 适合多目标优化问题，且需要基于客观数据对各目标进行赋权以平衡多指标 |
| 14 | 遗传算法（GA）+ 约束规划（CP） | GA 处理全局搜索，CP 强化约束满足能力，适合强约束优化问题（如调度） | 航班调度（含机组 / 机场约束）、课程表编排 | 适合存在大量且严格约束条件的复杂优化问题，尤其是调度类问题 |
| 15 | 模拟退火（SA）+ 神经网络（NN） | SA 优化 NN 的初始权重避免局部最优，NN 拟合复杂关系，适合预测 - 优化联动 | 股票价格预测与投资组合优化、气象参数优化 | 适合需要先进行复杂关系预测，再基于预测结果进行优化的联动问题 |

# 一、15种组合优化模型的详细解析

### 1. 遗传算法（GA）+ 模拟退火（SA）

* **基本原理**：GA 通过选择、交叉、变异模拟生物进化过程进行全局搜索；SA 模拟物理退火过程，以概率接受劣质解避免局部最优。二者结合时，GA 提供全局解空间探索，SA 对 GA 生成的较优解进行局部精细优化。
* **公式推导**：
  + GA 交叉操作：若父代染色体为 ，交叉后子代，（为交叉因子）。
  + SA 接受概率：，其中为能量差（目标函数差），为当前温度。
* **主要优势**：既保持 GA 的群体多样性，又借助 SA 的概率跳转能力跳出局部最优，提升解的精度和稳定性。
* **适合赛题**：复杂离散优化问题，如 “物流配送路径优化” “车间多机调度” 等含多约束、多局部最优的场景。

### 2. 粒子群优化（PSO）+ 禁忌搜索（TS）

* **基本原理**：PSO 中粒子通过跟踪个体极值和全局极值更新位置（）；TS 通过禁忌表记录已搜索解，避免重复迭代。组合时，PSO 快速定位全局较优区域，TS 在该区域内深度搜索。
* **公式推导**：
  + PSO 位置更新：（为惯性权重，c1，c2为学习因子）。
  + TS 禁忌长度：（为解空间维度），确保搜索多样性。
* **主要优势**：PSO 的快速收敛性与 TS 的局部寻优深度结合，适合高维离散问题。
* **适合赛题**：“电力系统负荷分配” “通信网络路由规划” 等需平衡效率与约束的问题。

### 3. 蚁群算法（ACO）+ 贪婪算法（Greedy）

* **基本原理**：ACO 通过信息素浓度引导蚂蚁搜索路径（）；Greedy 通过启发式规则（如最短距离优先）生成初始解。组合时，Greedy 加速初始解生成，ACO 迭代优化路径。
* **公式推导**：
  + ACO 转移概率：（为信息素与启发因子权重）。
  + Greedy 初始解：（为距离函数）。
* **主要优势**：减少 ACO 初期信息素匮乏导致的收敛缓慢问题，提升算法效率。
* **适合赛题**：“旅行商问题（TSP）” “无人机巡检路径规划” 等路径优化类赛题。

### 4. 支持向量机（SVM）+ 遗传算法（GA）

* **基本原理**：SVM 通过核函数（如 RBF 核）将数据映射到高维空间实现分类 / 回归；GA 优化 SVM 的核参数和惩罚因子。
* **公式推导**：
  + SVM 目标函数：（为松弛变量）。
  + GA 适应度函数：（以 SVM 分类准确率为优化目标）。
* **主要优势**：避免人为设置 SVM 参数导致的性能波动，提升模型泛化能力。
* **适合赛题**：“基于数据的工业参数优化” “故障诊断中的阈值确定” 等数据驱动类问题。

### 5. 模糊逻辑（FL）+ 粒子群优化（PSO）

* **基本原理**：FL 用隶属度函数（如三角形隶属度）处理模糊信息；PSO 优化模糊规则库或隶属度函数参数。
* **公式推导**：
  + 模糊推理：若是且是，则是，输出。
* PSO 优化目标：预测实际
* 其中 m 为样本数量，预测是 FL 的输出，实际是真实值。
* **主要优势**：处理含不确定性（如 “高负荷” “低能耗”）的优化问题，增强模型鲁棒性。
* **适合赛题**：“交通信号配时优化” “环境污染物减排决策” 等含模糊约束的场景。

### 6. 深度学习（DNN）+ 梯度下降（GD）

* **基本原理**：DNN 通过多层非线性变换（，为激活函数）拟合复杂关系；GD（或其变种如 Adam）优化网络参数b。
* **公式推导**：
  + 损失函数：（均方误差）。
  + GD 参数更新：，（为学习率）。
* **主要优势**：DNN 捕捉高维数据非线性特征，GD 高效优化大规模参数，适合大数据场景。
* **适合赛题**：“能源负荷预测与调度” “金融资产组合优化” 等含海量数据的问题。

### 7. 多目标遗传算法（MOGA）+ 层次分析法（AHP）

* **基本原理**：MOGA 通过非支配排序生成帕累托最优解集；AHP 将多目标（如成本、质量）转化为权重向量（，），筛选最优方案。
* **公式推导**：
  + MOGA 适应度分配：基于解的非支配等级和拥挤度，等级越低、拥挤度越大，适应度越高。
  + AHP 一致性检验：（为一致性指标，为随机指标）。
* **主要优势**：兼顾多目标优化的客观性与决策的主观性，平衡冲突目标（如成本与效率）。
* **适合赛题**：“工程方案选型” “供应链多目标优化” 等需权衡多个指标的赛题。

### 8. 模拟退火（SA）+ 线性规划（LP）

* **基本原理**：SA 处理非线性、离散变量；LP 对 SA 搜索过程中产生的线性子问题（）精确求解。
* **公式推导**：
  + SA 温度更新：（为降温系数）。
  + LP 对偶问题：（通过强对偶性验证解的最优性）。
* **主要优势**：SA 解决 LP 无法处理的非线性问题，LP 提升局部解的精确性，适合混合优化问题。
* **适合赛题**：“救灾物资分配” “混合整数规划问题” 等含线性与非线性子问题的场景。

### 9. 贝叶斯优化（BO）+ 随机森林（RF）

* **基本原理**：RF 构建代理模型预测目标函数值（，为决策树）；BO 基于高斯过程和采集函数（如 EI：）搜索最优解。
* **公式推导**：
  + RF 袋外误差：（评估模型精度）。
  + BO 后验分布：（为历史数据）。
* **主要优势**：减少昂贵实验（如材料测试）的次数，适合高成本优化问题。
* **适合赛题**：“材料配方优化” “工业设备参数调优” 等实验成本高的场景。

### 10. 禁忌搜索（TS）+ 0-1 整数规划（IP）

* **基本原理**：TS 通过禁忌表（）避免重复搜索；0-1 IP 对离散变量（）构建约束（）。组合时，TS 生成候选解，IP 验证解的可行性并优化。
* **公式推导**：
  + TS 邻域生成：ä¸Žåä¸ªå?˜é‡?（二进制变量翻转）。
  + 0-1 IP 目标函数：。
* **主要优势**：TS 提升 IP 的搜索效率，IP 确保解满足整数约束，适合离散组合优化。
* **适合赛题**：“仓库选址” “投资组合选择” 等含 0-1 决策变量的问题。

### 11. 人工蜂群算法（ABC）+ 差分进化（DE）

* **基本原理**：ABC 通过雇佣蜂、观察蜂、侦察蜂分工搜索（雇佣蜂位置更新：）；DE 通过差分变异（）和交叉操作优化。
* **公式推导**：
  + ABC 适应度函数：（为目标函数值，最小化问题）。
  + DE 交叉概率：（）。
* **主要优势**：ABC 的全局勘探与 DE 的局部开发结合，提升收敛速度和解的多样性。
* **适合赛题**：“机器人路径规划” “化工反应参数优化” 等非线性优化问题。

### 12. 模糊神经网络（FNN）+ 粒子群优化（PSO）

* **基本原理**：FNN 融合 FL 的模糊推理与 NN 的学习能力（网络输出）；PSO 优化网络权重和隶属度函数参数。
* **公式推导**：
  + FNN 误差函数：。
  + PSO 权重更新：，。
* **主要优势**：处理模糊信息与非线性关系，PSO 避免 FNN 梯度下降陷入局部最优。
* **适合赛题**：“污水处理参数优化” “医疗资源调度” 等含不确定输入的场景。

### 13. 多目标粒子群优化（MOPSO）+ 熵权法（EWM）

* **基本原理**：MOPSO 通过外部档案存储帕累托解，粒子跟踪档案中的全局最优；EWM 基于指标熵值（，）计算客观权重。
* **公式推导**：
  + MOPSO 拥挤距离：（维持解的多样性）。
  + EWM 权重：。
* **主要优势**：MOPSO 生成多目标解，EWM 避免主观赋权偏差，适合客观决策场景。
* **适合赛题**：“生态保护与经济发展平衡” “物流多目标优化（成本 / 碳排放）” 等问题。

### 14. 遗传算法（GA）+ 约束规划（CP）

* **基本原理**：GA 全局搜索解空间，CP 通过约束传播（如变量域缩减：）处理强约束。组合时，GA 提供候选解，CP 验证并修复约束违反。
* **公式推导**：
  + GA 变异操作：（为随机扰动，确保解的多样性）。
  + CP 约束满足：（通过回溯搜索验证解的可行性）。
* **主要优势**：GA 提升 CP 的搜索效率，CP 增强解对强约束（如时间窗、资源限制）的满足能力。
* **适合赛题**：“航班调度” “课程表编排” 等含复杂约束的优化问题。

### 15. 模拟退火（SA）+ 神经网络（NN）

* **基本原理**：NN 拟合目标函数（）；SA 优化 NN 初始权重，避免局部最优，并基于 NN 预测结果搜索最优解。
* **公式推导**：
  + SA 初始温度：（为初始接受概率）。
  + NN 反向传播：（为误差项）。
* **主要优势**：SA 优化 NN 的学习过程，NN 降低 SA 对真实函数的评估成本，适合预测 - 优化联动问题。
* **适合赛题**：“股票投资组合优化” “气象参数优化与灾害预警” 等需先预测后优化的场景。

# 二、15种组合优化模型的数学建模案例题目

### 1. 遗传算法（GA）+ 模拟退火（SA）组合模型案例题目

**题目：城市生鲜电商配送路径优化问题**

* **问题背景**：某生鲜电商企业在一线城市运营，需每日从 3 个仓库向 50 个社区站点配送生鲜产品。生鲜产品保质期短（常温下仅 12 小时），且运输过程中需维持 0-4℃冷链，配送成本高（含冷链能耗、超时损耗）。当前配送路线常因交通拥堵、重复绕路导致成本超支，客户投诉率达 15%。
* **问题描述**：需优化从各仓库到社区站点的配送路线，目标包括：① 最小化总配送时间（≤8 小时）；② 最小化总运输成本（含油费、冷链能耗）；③ 确保每个站点的生鲜损耗率≤5%。需考虑的约束：每个配送员日均工作时间≤8 小时，每辆车最大载重 500kg，早高峰（7:00-9:00）部分路段限行。
* **数据情况**：提供 3 个月的历史数据，包括：各仓库与站点的距离矩阵（单位：km）、各时段路段平均通行时间（含拥堵系数）、每辆车的冷链能耗率（元 /km）、不同超时时长对应的生鲜损耗率（如超时 1 小时损耗率增加 2%）、各站点的日订单量（10-80kg）。

### 1. 遗传算法（GA）+ 模拟退火（SA）求解城市生鲜电商配送路径优化代码

|  |
| --- |
| import numpy as np  import pandas as pd  import random  import math  from datetime import datetime, timedelta  import matplotlib.pyplot as plt  # 数据结构定义  class Warehouse:  def \_\_init\_\_(self, id, x, y):  self.id = id  self.x = x  self.y = y    class Customer:  def \_\_init\_\_(self, id, x, y, demand, time\_window\_start, time\_window\_end):  self.id = id  self.x = x  self.y = y  self.demand = demand # 需求量(kg)  self.time\_window\_start = time\_window\_start # 时间窗开始  self.time\_window\_end = time\_window\_end # 时间窗结束  class Vehicle:  def \_\_init\_\_(self, id, capacity=500, max\_working\_hours=8):  self.id = id  self.capacity = capacity # 最大载重(kg)  self.max\_working\_hours = max\_working\_hours # 最大工作时间(小时)  self.cost\_per\_km = 2.5 # 每公里成本(油费等)  self.cold\_energy\_per\_km = 1.2 # 每公里冷链能耗成本  # 生成模拟数据（实际应用中可替换为真实数据）  def generate\_sample\_data(num\_customers=50, num\_warehouses=3):  # 生成仓库  warehouses = []  for i in range(num\_warehouses):  warehouses.append(Warehouse(i, random.uniform(0, 100), random.uniform(0, 100)))    # 生成客户点（社区站点）  customers = []  for i in range(num\_customers):  # 随机位置  x = random.uniform(0, 100)  y = random.uniform(0, 100)  # 随机需求量(10-80kg)  demand = random.uniform(10, 80)  # 时间窗 (假设为9:00-18:00之间的随机时间窗)  start\_hour = random.randint(9, 15)  end\_hour = start\_hour + random.randint(2, 3)  customers.append(Customer(i, x, y, demand, start\_hour, end\_hour))    # 生成距离矩阵  locations = warehouses + customers  num\_locations = len(locations)  distance\_matrix = np.zeros((num\_locations, num\_locations))    for i in range(num\_locations):  for j in range(num\_locations):  if i != j:  # 计算欧氏距离  distance = math.hypot(locations[i].x - locations[j].x,  locations[i].y - locations[j].y)  distance\_matrix[i][j] = distance    # 生成时间矩阵（考虑拥堵因素）  time\_matrix = np.zeros((num\_locations, num\_locations))  for i in range(num\_locations):  for j in range(num\_locations):  if i != j:  # 基础时间（km / 30km/h）  base\_time = distance\_matrix[i][j] / 30  # 随机拥堵系数(0.8-1.5)  congestion\_factor = random.uniform(0.8, 1.5)  time\_matrix[i][j] = base\_time \* congestion\_factor    # 生成早高峰限行路段 (7:00-9:00)  restricted\_roads = set()  for \_ in range(int(num\_locations \* num\_locations \* 0.1)): # 10%的路段限行  i = random.randint(0, num\_locations-1)  j = random.randint(0, num\_locations-1)  if i != j:  restricted\_roads.add((i, j))    return warehouses, customers, distance\_matrix, time\_matrix, restricted\_roads  # 遗传算法实现  class GeneticAlgorithm:  def \_\_init\_\_(self, warehouses, customers, distance\_matrix, time\_matrix, restricted\_roads,  pop\_size=100, generations=100, mutation\_rate=0.02):  self.warehouses = warehouses  self.customers = customers  self.distance\_matrix = distance\_matrix  self.time\_matrix = time\_matrix  self.restricted\_roads = restricted\_roads  self.num\_warehouses = len(warehouses)  self.num\_customers = len(customers)  self.pop\_size = pop\_size  self.generations = generations  self.mutation\_rate = mutation\_rate    # 创建车辆（假设每仓库5辆车）  self.vehicles = []  for i in range(self.num\_warehouses \* 5):  self.vehicles.append(Vehicle(i))    # 所有位置（仓库+客户）  self.locations = warehouses + customers    # 初始化种群  def initialize\_population(self):  population = []  for \_ in range(self.pop\_size):  # 随机分配客户到仓库  customer\_assignments = [random.randint(0, self.num\_warehouses-1)  for \_ in range(self.num\_customers)]    # 为每个仓库生成路径  routes = []  for warehouse\_idx in range(self.num\_warehouses):  # 获取分配给该仓库的客户  assigned\_customers = [i for i, assign in enumerate(customer\_assignments)  if assign == warehouse\_idx]    # 随机排序客户，形成初始路径  if assigned\_customers:  random.shuffle(assigned\_customers)  # 路径格式: [仓库索引, 客户1, 客户2, ..., 仓库索引]  route = [warehouse\_idx] + assigned\_customers + [warehouse\_idx]  routes.append(route)    population.append(routes)  return population    # 计算路径的适应度  def calculate\_fitness(self, routes):  total\_distance = 0  total\_time = 0  total\_cost = 0  total\_penalty = 0  max\_loss\_rate = 0    for route in routes:  if len(route) < 3: # 至少包含仓库-客户-仓库  continue    # 计算该路径的总距离、时间和载重  route\_distance = 0  route\_time = 0  route\_load = 0    # 计算载重  for customer\_idx in route[1:-1]: # 排除首尾的仓库  route\_load += self.customers[customer\_idx].demand    # 检查载重约束  if route\_load > self.vehicles[0].capacity:  total\_penalty += (route\_load - self.vehicles[0].capacity) \* 10 # 超重惩罚    # 计算时间和距离  current\_time = 9.0 # 假设9点开始配送（避开早高峰）  prev\_location\_idx = route[0] # 起点仓库    for location\_idx in route[1:]:  # 转换为全局索引（仓库+客户）  from\_idx = prev\_location\_idx if prev\_location\_idx < self.num\_warehouses else \  self.num\_warehouses + prev\_location\_idx  to\_idx = location\_idx if location\_idx < self.num\_warehouses else \  self.num\_warehouses + location\_idx    # 累加距离  distance = self.distance\_matrix[from\_idx][to\_idx]  route\_distance += distance    # 累加时间（考虑是否在限行时段）  travel\_time = self.time\_matrix[from\_idx][to\_idx]  route\_time += travel\_time    # 检查是否经过限行路段且在限行时间内  if (from\_idx, to\_idx) in self.restricted\_roads and 7 <= current\_time < 9:  total\_penalty += 100 # 限行惩罚    # 更新当前时间  current\_time += travel\_time    # 如果是客户点，检查时间窗  if location\_idx >= self.num\_warehouses: # 是客户点  customer = self.customers[location\_idx - self.num\_warehouses]  # 时间窗惩罚  if current\_time < customer.time\_window\_start:  # 早到惩罚  total\_penalty += (customer.time\_window\_start - current\_time) \* 5  current\_time = customer.time\_window\_start # 等待到时间窗开始  elif current\_time > customer.time\_window\_end:  # 迟到惩罚，同时计算损耗率  delay = current\_time - customer.time\_window\_end  total\_penalty += delay \* 20  # 计算损耗率（超时1小时增加2%）  loss\_rate = delay \* 2  if loss\_rate > max\_loss\_rate:  max\_loss\_rate = loss\_rate    prev\_location\_idx = location\_idx    # 检查工作时间约束  if route\_time > self.vehicles[0].max\_working\_hours:  total\_penalty += (route\_time - self.vehicles[0].max\_working\_hours) \* 30 # 超时工作惩罚    # 累加总距离、时间  total\_distance += route\_distance  total\_time += route\_time    # 计算成本（油费+冷链能耗）  vehicle\_cost = route\_distance \* (self.vehicles[0].cost\_per\_km +  self.vehicles[0].cold\_energy\_per\_km)  total\_cost += vehicle\_cost    # 损耗率惩罚（超过5%）  if max\_loss\_rate > 5:  total\_penalty += (max\_loss\_rate - 5) \* 50    # 多目标综合适应度（将所有目标转换为最小化问题）  # 权重可以根据实际业务需求调整  fitness = (total\_time \* 10) + (total\_cost \* 0.01) + (total\_penalty)  return fitness    # 选择操作（轮盘赌选择）  def select\_parents(self, population, fitnesses):  # 适应度越小越好，转换为选择概率  min\_fitness = min(fitnesses)  max\_fitness = max(fitnesses)    # 转换为越大越好  adjusted\_fitness = [max\_fitness - f + min\_fitness for f in fitnesses]  total\_adjusted = sum(adjusted\_fitness)    if total\_adjusted == 0: # 所有适应度相同  return random.sample(population, 2)    # 计算选择概率  probabilities = [f / total\_adjusted for f in adjusted\_fitness]    # 选择两个父代  parents = random.choices(population, probabilities, k=2)  return parents    # 交叉操作  def crossover(self, parent1, parent2):  # 简单的交叉策略：随机选择父代的部分路径  if random.random() < 0.7: # 70%的概率进行交叉  # 随机选择交叉点  crossover\_point = random.randint(0, min(len(parent1), len(parent2)) - 1)  child = parent1[:crossover\_point] + parent2[crossover\_point:]    # 简单去重处理（实际应用中可能需要更复杂的策略）  seen = set()  unique\_child = []  for route in child:  if tuple(route) not in seen:  seen.add(tuple(route))  unique\_child.append(route)  return unique\_child  else:  # 不交叉，随机返回一个父代  return random.choice([parent1, parent2])    # 变异操作  def mutate(self, individual):  mutated = [route.copy() for route in individual]    for i in range(len(mutated)):  if random.random() < self.mutation\_rate and len(mutated[i]) > 3:  # 对路径中的客户点进行随机交换（变异）  idx1 = random.randint(1, len(mutated[i])-2)  idx2 = random.randint(1, len(mutated[i])-2)  mutated[i][idx1], mutated[i][idx2] = mutated[i][idx2], mutated[i][idx1]    # 随机重新分配部分客户  if random.random() < self.mutation\_rate:  if len(mutated) > 1 and any(len(route) > 3 for route in mutated):  # 选择两个不同的路径  route\_idx1, route\_idx2 = random.sample(range(len(mutated)), 2)  route1 = mutated[route\_idx1]  route2 = mutated[route\_idx2]    if len(route1) > 3 and len(route2) > 3:  # 从第一个路径中选择一个客户移到第二个路径  customer\_idx = random.randint(1, len(route1)-2)  customer = route1.pop(customer\_idx)  insert\_pos = random.randint(1, len(route2)-1)  route2.insert(insert\_pos, customer)    return mutated    # 运行遗传算法  def run(self):  # 初始化种群  population = self.initialize\_population()  best\_fitness = float('inf')  best\_solution = None    # 记录进化过程  fitness\_history = []    for gen in range(self.generations):  # 计算适应度  fitnesses = [self.calculate\_fitness(individual) for individual in population]    # 记录最佳解  current\_best\_idx = np.argmin(fitnesses)  current\_best\_fitness = fitnesses[current\_best\_idx]  current\_best\_solution = population[current\_best\_idx]    if current\_best\_fitness < best\_fitness:  best\_fitness = current\_best\_fitness  best\_solution = current\_best\_solution    fitness\_history.append(current\_best\_fitness)    # 打印进度  if gen % 10 == 0:  print(f"Generation {gen}: Best Fitness = {current\_best\_fitness:.2f}")    # 创建下一代  new\_population = []    # 保留精英  elite\_count = int(self.pop\_size \* 0.1)  elite\_indices = np.argsort(fitnesses)[:elite\_count]  new\_population.extend([population[i] for i in elite\_indices])    # 生成剩余个体  while len(new\_population) < self.pop\_size:  # 选择父母  parent1, parent2 = self.select\_parents(population, fitnesses)    # 交叉  child = self.crossover(parent1, parent2)    # 变异  child = self.mutate(child)    new\_population.append(child)    population = new\_population    # 绘制适应度进化曲线  plt.figure(figsize=(10, 6))  plt.plot(fitness\_history)  plt.title('Evolution of Best Fitness')  plt.xlabel('Generation')  plt.ylabel('Best Fitness')  plt.grid(True)  plt.show()    return best\_solution, best\_fitness  # 评估并输出最佳方案  def evaluate\_solution(solution, warehouses, customers, distance\_matrix, time\_matrix, vehicles):  if not solution:  print("No solution found!")  return    total\_distance = 0  total\_time = 0  total\_cost = 0  max\_loss\_rate = 0  vehicle\_used = 0    print("\n=== 配送路径优化结果 ===")    for route\_idx, route in enumerate(solution):  if len(route) < 3:  continue    vehicle\_used += 1  warehouse\_id = route[0]  route\_distance = 0  route\_time = 0  route\_load = 0    # 计算载重  for customer\_idx in route[1:-1]:  route\_load += customers[customer\_idx].demand    # 计算距离和时间  current\_time = 9.0 # 假设9点开始配送  prev\_location\_idx = route[0]    print(f"\n车辆 {vehicle\_used} (从仓库 {warehouse\_id} 出发):")  print(f"总载重: {route\_load:.2f} kg")    for location\_idx in route[1:]:  # 转换为全局索引  from\_idx = prev\_location\_idx if prev\_location\_idx < len(warehouses) else \  len(warehouses) + prev\_location\_idx  to\_idx = location\_idx if location\_idx < len(warehouses) else \  len(warehouses) + location\_idx    distance = distance\_matrix[from\_idx][to\_idx]  travel\_time = time\_matrix[from\_idx][to\_idx]    route\_distance += distance  route\_time += travel\_time  current\_time += travel\_time    if location\_idx < len(warehouses): # 仓库  print(f"返回仓库 {location\_idx}, 到达时间: {current\_time:.2f} 时, "  f"距离: {distance:.2f} km, 耗时: {travel\_time:.2f} 小时")  else: # 客户点  customer = customers[location\_idx - len(warehouses)]  # 计算损耗率  delay = max(0, current\_time - customer.time\_window\_end)  loss\_rate = delay \* 2  if loss\_rate > max\_loss\_rate:  max\_loss\_rate = loss\_rate    status = "准时" if current\_time <= customer.time\_window\_end else f"迟到 {delay:.2f} 小时"  print(f"配送至站点 {location\_idx - len(warehouses)}, 到达时间: {current\_time:.2f} 时, "  f"时间窗: [{customer.time\_window\_start}, {customer.time\_window\_end}], "  f"{status}, 损耗率: {loss\_rate:.2f}%, "  f"距离: {distance:.2f} km, 耗时: {travel\_time:.2f} 小时")    prev\_location\_idx = location\_idx    # 计算成本  vehicle\_cost = route\_distance \* (vehicles[0].cost\_per\_km + vehicles[0].cold\_energy\_per\_km)    print(f"该路径总距离: {route\_distance:.2f} km")  print(f"该路径总时间: {route\_time:.2f} 小时")  print(f"该路径总成本: {vehicle\_cost:.2f} 元")    total\_distance += route\_distance  total\_time += route\_time  total\_cost += vehicle\_cost    print("\n=== 汇总信息 ===")  print(f"使用车辆数: {vehicle\_used}")  print(f"总配送距离: {total\_distance:.2f} km")  print(f"总配送时间: {total\_time:.2f} 小时")  print(f"总配送成本: {total\_cost:.2f} 元")  print(f"最大损耗率: {max\_loss\_rate:.2f}%")  print(f"是否满足时间约束: {'是' if total\_time <= 8 else '否'}")  print(f"是否满足损耗约束: {'是' if max\_loss\_rate <= 5 else '否'}")  # 主函数  def main():  # 生成模拟数据  print("生成模拟数据...")  warehouses, customers, distance\_matrix, time\_matrix, restricted\_roads = generate\_sample\_data(50, 3)    # 创建遗传算法实例  ga = GeneticAlgorithm(  warehouses,  customers,  distance\_matrix,  time\_matrix,  restricted\_roads,  pop\_size=80,  generations=100,  mutation\_rate=0.05  )    # 运行算法  print("开始优化配送路径...")  best\_solution, best\_fitness = ga.run()    # 评估并输出结果  evaluate\_solution(best\_solution, warehouses, customers, distance\_matrix, time\_matrix, ga.vehicles)  if \_\_name\_\_ == "\_\_main\_\_":  main() |

### 2. 粒子群优化（PSO）+ 禁忌搜索（TS）组合模型案例题目

**题目：半导体车间多设备协同调度问题**

* **问题背景**：某半导体工厂的芯片封装车间有 10 台不同型号的封装设备，需处理 80 批不同规格的芯片（每批数量 500-2000 片）。芯片封装需经过 “固晶 - 焊线 - 塑封 - 切筋” 4 道工序，不同设备的工序加工效率差异显著，且设备切换规格时需停机调试（耗时 30-60 分钟）。当前调度方案导致设备闲置率达 25%，订单平均延期率 18%。
* **问题描述**：需优化各批次芯片在设备间的加工顺序和时间分配，目标包括：① 最小化总生产周期（≤72 小时）；② 最大化设备利用率；③ 最小化订单延期时间（重要订单延期罚款是普通订单的 3 倍）。约束条件：每道工序必须按顺序执行，某设备同一时间只能处理一批芯片，部分精密设备每日需预留 2 小时维护时间。
* **数据情况**：提供各批次芯片的工序需求（如 A 批次需在设备 1 完成固晶、设备 5 完成焊线）、各设备的工序加工速率（片 / 分钟）、设备切换规格的调试时间矩阵、各订单的交货截止时间及优先级标签、设备每日可用时间窗口。

### 2. 粒子群优化（PSO）+ 禁忌搜索（TS）求解半导体车间调度代码

|  |
| --- |
| import numpy as np  import random  import matplotlib.pyplot as plt  from matplotlib.patches import Patch  import matplotlib.font\_manager as fm  # 设置中文显示 - 尝试多种常见中文字体，增加兼容性  def setup\_chinese\_fonts():  # 尝试的中文字体列表，涵盖不同系统  chinese\_fonts = [  "SimHei", "WenQuanYi Micro Hei", "Heiti TC", # 常见中文字体  "Microsoft YaHei", "SimSun", "KaiTi", # Windows系统  "Arial Unicode MS", # Mac系统  "WenQuanYi Zen Hei" # Linux系统  ]    # 查找系统中可用的中文字体  available\_fonts = [f for f in fm.findSystemFonts() if any(font in f for font in chinese\_fonts)]    if available\_fonts:  # 使用找到的第一个中文字体  font\_name = fm.FontProperties(fname=available\_fonts[0]).get\_name()  plt.rcParams["font.family"] = font\_name  else:  # 如果没有找到中文字体，使用默认字体并给出警告  print("警告: 未找到中文字体，可能无法正常显示中文标签")    # 正确显示负号  plt.rcParams["axes.unicode\_minus"] = False  # 初始化中文字体  setup\_chinese\_fonts()  # 数据初始化  np.random.seed(42)  n\_jobs = 80 # 80批芯片  n\_machines = 10 # 10台设备  n\_stages = 4 # 4道工序  # 每批芯片的工序需求（每道工序对应一台设备）  processing\_machines = np.random.randint(0, n\_machines, size=(n\_jobs, n\_stages))  # 每台设备的工序加工速率（片/分钟）  process\_rate = np.random.uniform(50, 200, size=(n\_machines, n\_stages))  # 设备切换规格的调试时间（分钟）  setup\_time = np.random.randint(30, 61, size=(n\_machines, n\_jobs, n\_jobs))  # 订单截止时间（小时）  due\_dates = np.random.uniform(24, 72, size=n\_jobs)  # 订单优先级（1-3，3为最高）  priority = np.random.randint(1, 4, size=n\_jobs)  # 每批数量（片）  batch\_sizes = np.random.randint(500, 2001, size=n\_jobs)  # 计算加工时间（分钟）  processing\_time = np.zeros((n\_jobs, n\_stages))  for j in range(n\_jobs):  for s in range(n\_stages):  machine = processing\_machines[j, s]  processing\_time[j, s] = batch\_sizes[j] / process\_rate[machine, s]  # 优化后的PSO参数  n\_particles = 15  max\_iter = 50  w = 0.7  c1, c2 = 0.6, 0.3  # 优化后的禁忌搜索参数  tabu\_size = 15  tabu\_list = []  # 粒子初始化 - 返回列表而不是numpy数组  def init\_particles(n\_particles, n\_jobs):  particles = []  for \_ in range(n\_particles):  if \_ % 3 == 0:  # 按截止时间排序的贪婪初始化  sorted\_indices = np.argsort(due\_dates)  particles.append(sorted\_indices.tolist()) # 转换为列表  else:  particles.append(random.sample(range(n\_jobs), n\_jobs))  return particles # 返回列表的列表，而非numpy数组  # 优化的makespan计算  def calculate\_makespan(sequence):  machine\_end = np.zeros((n\_machines, n\_stages))  job\_end = np.zeros((n\_jobs, n\_stages))  last\_job\_on\_machine = {m: {s: -1 for s in range(n\_stages)} for m in range(n\_machines)}    for job in sequence:  for stage in range(n\_stages):  machine = processing\_machines[job, stage]  prev\_stage\_end = job\_end[job, stage-1] if stage > 0 else 0  current\_machine\_end = machine\_end[machine, stage]  start\_time = max(prev\_stage\_end, current\_machine\_end)    last\_job = last\_job\_on\_machine[machine][stage]  if last\_job != -1:  start\_time += setup\_time[machine, last\_job, job]    job\_end[job, stage] = start\_time + processing\_time[job, stage]  machine\_end[machine, stage] = job\_end[job, stage]  last\_job\_on\_machine[machine][stage] = job    makespan = np.max(job\_end)  return makespan, job\_end, machine\_end  # 合并计算适应度所需的所有指标  def calculate\_fitness\_metrics(sequence):  makespan, job\_end, machine\_end = calculate\_makespan(sequence)    total\_tardiness = 0  for job in sequence:  job\_tardiness = max(0, makespan/60 - due\_dates[job])  total\_tardiness += job\_tardiness \* priority[job]    total\_processing = np.sum(processing\_time)  total\_machine\_time = np.sum(machine\_end)  machine\_util = total\_processing / total\_machine\_time if total\_machine\_time > 0 else 0    return makespan, total\_tardiness, machine\_util  # 优化的适应度函数  def fitness(sequence):  makespan, tardiness, machine\_util = calculate\_fitness\_metrics(sequence)  return 1 / (makespan + tardiness \* 60) + machine\_util \* 0.1  # 优化的禁忌搜索  def tabu\_search(best\_seq, fitness\_func, max\_steps=20):  current\_seq = best\_seq.copy()  current\_fit = fitness\_func(current\_seq)  best\_tabu\_seq = current\_seq  best\_tabu\_fit = current\_fit    for step in range(max\_steps):  # 多样化邻域生成  if random.random() < 0.7: # 交换  a, b = random.sample(range(len(current\_seq)), 2)  new\_seq = current\_seq.copy()  new\_seq[a], new\_seq[b] = new\_seq[b], new\_seq[a]  elif random.random() < 0.85: # 插入  a, b = random.sample(range(len(current\_seq)), 2)  new\_seq = current\_seq.copy()  val = new\_seq.pop(a) # 现在可以安全使用pop方法  new\_seq.insert(b, val)  else: # 反转子序列  a, b = sorted(random.sample(range(len(current\_seq)), 2))  new\_seq = current\_seq.copy()  new\_seq[a:b+1] = new\_seq[a:b+1][::-1]    # 检查禁忌表  seq\_tuple = tuple(new\_seq)  if seq\_tuple in tabu\_list:  continue    new\_fit = fitness\_func(new\_seq)  if new\_fit > best\_tabu\_fit:  best\_tabu\_seq = new\_seq  best\_tabu\_fit = new\_fit  current\_seq = new\_seq  current\_fit = new\_fit  tabu\_list.append(seq\_tuple)  if len(tabu\_list) > tabu\_size:  tabu\_list.pop(0)  elif new\_fit > current\_fit:  current\_seq = new\_seq  current\_fit = new\_fit    return best\_tabu\_seq, best\_tabu\_fit  # 优化的PSO主算法  def pso\_ts\_optimize():  # 粒子现在是列表的列表，而非numpy数组  particles = init\_particles(n\_particles, n\_jobs)    # 初始化速度  velocities = []  for \_ in range(n\_particles):  vel = [set() for \_ in range(n\_jobs)]  for \_\_ in range(n\_jobs//4):  a, b = random.sample(range(n\_jobs), 2)  vel[a].add(b)  vel[b].add(a)  velocities.append(vel)    # 初始化最优解  pbest = [p.copy() for p in particles] # 列表复制  pbest\_fitness = [fitness(seq) for seq in particles]  gbest\_idx = np.argmax(pbest\_fitness)  gbest = pbest[gbest\_idx].copy()  gbest\_fitness = pbest\_fitness[gbest\_idx]    # 禁忌搜索优化初始全局最优  gbest, gbest\_fitness = tabu\_search(gbest, fitness)    # 记录优化过程  best\_fitness\_history = [gbest\_fitness]    for iter in range(max\_iter):  current\_w = w \* (1 - iter/max\_iter)    for i in range(n\_particles):  # 更新速度  new\_vel = [set() for \_ in range(n\_jobs)]    # 保留部分惯性速度  for pos in range(n\_jobs):  if random.random() < current\_w and velocities[i][pos]:  new\_vel[pos] = set(velocities[i][pos])    # 向个体最优学习  if random.random() < c1:  diff\_pos = [pos for pos in range(n\_jobs) if particles[i][pos] != pbest[i][pos]]  if diff\_pos:  a, b = random.sample(diff\_pos, 2) if len(diff\_pos)>=2 else (diff\_pos[0], random.randint(0, n\_jobs-1))  new\_vel[a].add(b)  new\_vel[b].add(a)    # 向全局最优学习  if random.random() < c2:  diff\_pos = [pos for pos in range(n\_jobs) if particles[i][pos] != gbest[pos]]  if diff\_pos:  a, b = random.sample(diff\_pos, 2) if len(diff\_pos)>=2 else (diff\_pos[0], random.randint(0, n\_jobs-1))  new\_vel[a].add(b)  new\_vel[b].add(a)    velocities[i] = new\_vel    # 更新位置（应用交换）  new\_seq = particles[i].copy()  swapped = set()  for a in range(n\_jobs):  if a not in swapped and velocities[i][a]:  b = random.choice(list(velocities[i][a]))  if b not in swapped:  new\_seq[a], new\_seq[b] = new\_seq[b], new\_seq[a]  swapped.add(a)  swapped.add(b)    particles[i] = new\_seq    # 评估适应度并更新最优解  current\_fit = fitness(new\_seq)  if current\_fit > pbest\_fitness[i]:  pbest[i] = new\_seq.copy()  pbest\_fitness[i] = current\_fit  if current\_fit > gbest\_fitness:  gbest = new\_seq.copy()  gbest\_fitness = current\_fit  if iter % 10 == 0:  gbest, gbest\_fitness = tabu\_search(gbest, fitness)    best\_fitness\_history.append(gbest\_fitness)    if iter % 5 == 0:  print(f"迭代 {iter}, 最优适应度: {gbest\_fitness:.4f}")    return gbest, gbest\_fitness, best\_fitness\_history  # 绘制甘特图  def plot\_gantt\_chart(sequence, n\_display=10):  makespan, job\_end, \_ = calculate\_makespan(sequence)    plt.figure(figsize=(15, 10))  colors = plt.cm.tab20(np.linspace(0, 1, n\_display))    for i, job in enumerate(sequence[:n\_display]):  for stage in range(n\_stages):  machine = processing\_machines[job, stage]  proc\_time = processing\_time[job, stage]  end\_time = job\_end[job, stage]  start\_time = end\_time - proc\_time    plt.barh(  y=machine \* n\_stages + stage,  width=proc\_time,  left=start\_time,  height=0.8,  color=colors[i],  edgecolor='black'  )    plt.xlabel("时间（分钟）")  plt.ylabel("设备-工序")  plt.title(f"半导体车间调度甘特图（前{n\_display}个作业）")    y\_ticks = []  y\_labels = []  for machine in range(n\_machines):  for stage in range(n\_stages):  y\_ticks.append(machine \* n\_stages + stage)  y\_labels.append(f"设备{machine}-工序{stage}")  plt.yticks(y\_ticks, y\_labels)    legend\_elements = [Patch(color=colors[i], label=f"作业{sequence[i]}") for i in range(n\_display)]  plt.legend(handles=legend\_elements, bbox\_to\_anchor=(1.05, 1), loc='upper left')    plt.tight\_layout()  return plt  # 绘制适应度进化曲线  def plot\_fitness\_curve(history):  plt.figure(figsize=(10, 6))  plt.plot(history)  plt.xlabel("迭代次数")  plt.ylabel("最优适应度")  plt.title("算法优化过程中的适应度进化曲线")  plt.grid(True, linestyle='--', alpha=0.7)  plt.tight\_layout()  return plt  # 运行算法  if \_\_name\_\_ == "\_\_main\_\_":  print("开始优化半导体车间调度方案...")  best\_sequence, best\_fit, fitness\_history = pso\_ts\_optimize()    # 计算优化结果指标  makespan, tardiness, \_ = calculate\_fitness\_metrics(best\_sequence)    # 输出结果  print("\n优化结果：")  print("最优加工顺序（作业索引）：", best\_sequence[:10], "...（共80个）")  print(f"总生产周期（分钟）：{makespan:.2f}")  print(f"总生产周期（小时）：{makespan/60:.2f}")  print(f"总加权延期时间（小时）：{tardiness:.2f}")    # 绘制甘特图  gantt\_plt = plot\_gantt\_chart(best\_sequence, n\_display=10)  gantt\_plt.show()    # 绘制适应度进化曲线  fitness\_plt = plot\_fitness\_curve(fitness\_history)  fitness\_plt.show() |

### 3. 蚁群算法（ACO）+ 贪婪算法（Greedy）组合模型案例题目

**题目：灾后应急物资多点配送路径规划问题**

* **问题背景**：某地区发生地震后，20 个乡镇成为受灾点，急需食品、药品、帐篷等物资。现有 3 个应急仓库（分别位于县城东、南、西部），仓库物资储备量有限，且部分道路因塌方仅能单向通行或限时通行（如上午 8:00-12:00 可通行）。救援车辆需在 72 小时内完成首轮物资投放，否则受灾点将出现物资短缺。
* **问题描述**：需规划救援车辆的配送路径，目标包括：① 最小化所有受灾点的物资送达时间差（避免个别点严重滞后）；② 最小化总运输里程；③ 确保每个受灾点的核心物资（食品、药品）覆盖率≥90%。约束条件：每辆车最大载重 8 吨，单次运输时间≤8 小时（含卸货），塌方路段需绕行且通行速度≤20km/h。
* **数据情况**：提供受灾点的位置坐标、各物资需求量（如甲乡镇需食品 2 吨、药品 50 箱）、仓库物资库存量、道路拓扑图（含双向 / 单向标识、通行时段、长度及正常路况下的行驶速度）、车辆平均卸货时间（0.5 小时 / 点）。

### 3. 蚁群算法（ACO）+ 贪婪算法（Greedy）求解灾后物资配送代码

|  |
| --- |
| import numpy as np  import random  import matplotlib.pyplot as plt  from math import sqrt  # 数据初始化  np.random.seed(42)  # 3个应急仓库坐标  warehouses = np.array([[0, 0], [50, 100], [100, 50]])  # 20个受灾点坐标  victims = np.random.rand(20, 2) \* 100  all\_points = np.vstack((warehouses, victims))  n\_victims = len(victims)  n\_warehouses = len(warehouses)  # 受灾点需求（食品吨、药品箱、帐篷顶）  demand = np.random.randint(1, 6, size=(n\_victims, 3)) # 每种物资1-5单位  # 仓库库存（食品吨、药品箱、帐篷顶）  stock = np.array([[20, 50, 30], [15, 40, 25], [18, 35, 20]])  # 道路通行时间（分钟）- 包含塌方路段修正  road\_time = np.zeros((len(all\_points), len(all\_points)))  for i in range(len(all\_points)):  for j in range(len(all\_points)):  dist = sqrt(((all\_points[i] - all\_points[j])\*\*2).sum())  # 10%概率为塌方路段，通行时间翻倍  if random.random() < 0.1:  road\_time[i][j] = dist \* 2  else:  road\_time[i][j] = dist  # 贪婪算法生成初始解  def greedy\_route(warehouse\_idx):  route = [warehouse\_idx]  visited = set([warehouse\_idx])  current = warehouse\_idx  remaining = set(range(n\_warehouses, n\_warehouses + n\_victims))    # 优先满足核心物资需求高的点  priority = np.sum(demand[:, :2], axis=1) # 食品+药品为核心物资  priority\_idx = np.argsort(-priority) + n\_warehouses # 转换为全局索引    for idx in priority\_idx:  if idx not in visited:  route.append(idx)  visited.add(idx)  current = idx    return route  # 蚁群算法参数  n\_ants = 30  iterations = 100  alpha = 1.0 # 信息素重要度  beta = 2.0 # 启发式信息重要度  rho = 0.1 # 信息素蒸发率  Q = 100 # 信息素增量常数  # 启发式信息（1/时间）  def heuristic\_info():  eta = np.zeros\_like(road\_time)  for i in range(len(road\_time)):  for j in range(len(road\_time)):  if i != j and road\_time[i][j] > 0:  eta[i][j] = 1.0 / road\_time[i][j]  return eta  # 适应度函数（综合时间、覆盖率、里程）  def fitness(route, warehouse\_idx):  total\_time = 0  total\_dist = 0  current\_stock = stock[warehouse\_idx].copy()  covered = 0 # 核心物资覆盖率    for i in range(len(route)-1):  u = route[i]  v = route[i+1]  total\_time += road\_time[u][v]  total\_dist += sqrt(((all\_points[u] - all\_points[v])\*\*2).sum())    # 计算物资覆盖率（仅对受灾点）  if v >= n\_warehouses:  vid = v - n\_warehouses # 受灾点索引  # 能满足的核心物资比例  food\_ratio = min(1.0, current\_stock[0] / demand[vid][0])  med\_ratio = min(1.0, current\_stock[1] / demand[vid][1])  covered += (food\_ratio + med\_ratio) / 2  # 消耗库存  current\_stock -= demand[vid]  current\_stock = np.maximum(current\_stock, 0) # 不能为负    # 平均覆盖率  avg\_coverage = covered / (len(route) - 1) # 减去仓库  # 时间惩罚（超过72小时=4320分钟）  time\_penalty = max(0, total\_time - 4320) \* 0.1  # 目标函数：最大化覆盖率，最小化时间和距离  return avg\_coverage / (total\_time + total\_dist + time\_penalty + 1e-6)  # 蚁群算法主函数  def aco\_greedy\_optimize():  best\_routes = []  eta = heuristic\_info() # 启发式信息    for warehouse in range(n\_warehouses):  # 初始化信息素  tau = np.ones\_like(road\_time) \* 0.1  # 贪婪算法生成初始最优解  best\_route = greedy\_route(warehouse)  best\_fitness = fitness(best\_route, warehouse)    for iter in range(iterations):  ant\_routes = []  ant\_fitnesses = []    for ant in range(n\_ants):  # 构建路径  route = [warehouse]  visited = set([warehouse])    while len(visited) < n\_victims + 1: # 所有点+仓库  current = route[-1]  # 可选节点  candidates = [v for v in range(len(all\_points))  if v not in visited and v != current]  if not candidates:  break    # 计算转移概率  probs = []  for v in candidates:  probs.append(tau[current][v]\*\* alpha \* eta[current][v] \*\* beta)  probs = np.array(probs) / np.sum(probs)    # 选择下一个节点  next\_v = np.random.choice(candidates, p=probs)  route.append(next\_v)  visited.add(next\_v)    ant\_routes.append(route)  ant\_fitnesses.append(fitness(route, warehouse))    # 更新最优解  current\_best\_idx = np.argmax(ant\_fitnesses)  if ant\_fitnesses[current\_best\_idx] > best\_fitness:  best\_route = ant\_routes[current\_best\_idx]  best\_fitness = ant\_fitnesses[current\_best\_idx]    # 信息素蒸发  tau = (1 - rho) \* tau    # 信息素更新  for i in range(n\_ants):  route = ant\_routes[i]  fit = ant\_fitnesses[i]  for j in range(len(route)-1):  u = route[j]  v = route[j+1]  tau[u][v] += Q \* fit # 根据适应度增加信息素    best\_routes.append((best\_route, best\_fitness, warehouse))    # 选择全局最优解  best\_idx = np.argmax([f for (r, f, w) in best\_routes])  return best\_routes[best\_idx]  # 运行并可视化  if \_\_name\_\_ == "\_\_main\_\_":  best\_route, best\_fit, warehouse = aco\_greedy\_optimize()  print(f"最优配送路线（从仓库{warehouse}出发）：", best\_route[:5], "...（省略部分点）")  print("最优适应度值：", best\_fit)    # 绘制路线图  plt.figure(figsize=(10, 8))  plt.scatter(warehouses[:,0], warehouses[:,1], c='red', s=100, label='应急仓库')  plt.scatter(victims[:,0], victims[:,1], c='blue', s=50, label='受灾点')  # 标记出发仓库  plt.scatter(all\_points[warehouse,0], all\_points[warehouse,1], c='green', s=150, label='出发仓库')  # 绘制路线  route\_points = all\_points[best\_route]  plt.plot(route\_points[:,0], route\_points[:,1], 'g-', linewidth=2)  plt.title('灾后物资最优配送路线')  plt.legend()  plt.show() |

### 4. 支持向量机（SVM）+ 遗传算法（GA）组合模型案例题目

**题目：锂电池生产参数优化与质量预测问题**

* **问题背景**：某锂电池工厂的电芯生产中，“匀浆 - 涂布 - 辊压” 工序的参数设置直接影响电芯容量和循环寿命。当前采用经验参数（如浆料固含量 50%、涂布速度 3m/min），导致电芯合格率仅 82%，且不同批次的容量偏差达 15%。工厂希望通过数据驱动方法优化参数，提升产品一致性。
* **问题描述**：需构建参数优化模型，目标包括：① 最大化电芯容量（≥3.6Ah）；② 最大化循环寿命（≥1000 次）；③ 最小化参数调整成本（如固含量每变化 1% 成本增加 200 元 / 批次）。需通过历史数据训练模型，找到最优参数组合（固含量 45%-55%、涂布速度 2-4m/min、辊压压力 10-20MPa 等）。
* **数据情况**：提供 1000 批次的生产数据，包括 5 个关键工艺参数的具体值、对应批次的电芯容量检测值、循环寿命测试结果、参数调整的成本记录，以及部分不合格品的缺陷类型（如析锂、鼓包）。

### 4. 支持向量机（SVM）+ 遗传算法（GA）求解锂电池生产参数优化代码

|  |
| --- |
| import numpy as np  import random  import matplotlib.pyplot as plt  from sklearn.svm import SVR  from sklearn.preprocessing import MinMaxScaler  from sklearn.model\_selection import train\_test\_split  from sklearn.metrics import mean\_squared\_error  import time  # 数据生成与预处理（模拟1000批次生产数据）  np.random.seed(42)  n\_samples = 1000  # 5个关键工艺参数及其范围  # [固含量(%)、涂布速度(m/min)、辊压压力(MPa)、温度(℃)、时间(min)]  param\_ranges = np.array([[45, 55], [2, 4], [10, 20], [80, 120], [30, 60]])  X = np.random.uniform(  low=param\_ranges[:, 0],  high=param\_ranges[:, 1],  size=(n\_samples, 5)  )  # 基准参数（经验值）  base\_params = np.array([50, 3, 15, 100, 45])  # 生成电芯容量（3.2-3.8Ah）- 模拟非线性关系  capacity = 3.2 + 0.02\*(X[:,0]-45) + 0.08\*(X[:,1]-2) + 0.03\*(X[:,2]-10) + \  0.006\*(X[:,3]-80) + 0.012\*(X[:,4]-30) - 0.002\*(X[:,0]-50)\*\*2 - \  0.01\*(X[:,1]-3)\*\* 2 + np.random.normal(0, 0.025, n\_samples)  capacity = np.clip(capacity, 3.2, 3.8)  # 生成循环寿命（800-1200次）  cycle\_life = 800 + 8\*(X[:,0]-45) + 40\*(X[:,1]-2) + 15\*(X[:,2]-10) + \  3\*(X[:,3]-80) + 4\*(X[:,4]-30) - 2\*(X[:,2]-15)\*\*2 + \  np.random.normal(0, 8, n\_samples)  cycle\_life = np.clip(cycle\_life, 800, 1200)  # 计算参数调整成本（与基准参数的偏差）  # 固含量每变化1%成本+200元，涂布速度每变化0.1m/min+150元，其余参数按比例计算  cost = 200 \* np.abs(X[:,0] - base\_params[0]) + \  1500 \* np.abs(X[:,1] - base\_params[1]) + \  100 \* np.abs(X[:,2] - base\_params[2]) + \  50 \* np.abs(X[:,3] - base\_params[3]) + \  30 \* np.abs(X[:,4] - base\_params[4])  # 数据归一化  scaler\_X = MinMaxScaler()  scaler\_cap = MinMaxScaler()  scaler\_life = MinMaxScaler()  X\_scaled = scaler\_X.fit\_transform(X)  capacity\_scaled = scaler\_cap.fit\_transform(capacity.reshape(-1, 1)).flatten()  life\_scaled = scaler\_life.fit\_transform(cycle\_life.reshape(-1, 1)).flatten()  # 划分训练集和测试集  X\_train, X\_test, cap\_train, cap\_test = train\_test\_split(  X\_scaled, capacity\_scaled, test\_size=0.2, random\_state=42)  \_, \_, life\_train, life\_test = train\_test\_split(  X\_scaled, life\_scaled, test\_size=0.2, random\_state=42)  # 定义SVM模型训练函数  def train\_svm\_models():  # 训练容量预测模型  svm\_cap = SVR(C=10, gamma=0.1, kernel='rbf')  svm\_cap.fit(X\_train, cap\_train)  cap\_pred = svm\_cap.predict(X\_test)  print(f"容量预测模型MSE: {mean\_squared\_error(cap\_test, cap\_pred):.6f}")    # 训练寿命预测模型  svm\_life = SVR(C=10, gamma=0.1, kernel='rbf')  svm\_life.fit(X\_train, life\_train)  life\_pred = svm\_life.predict(X\_test)  print(f"寿命预测模型MSE: {mean\_squared\_error(life\_test, life\_pred):.6f}")    return svm\_cap, svm\_life  # 遗传算法优化生产参数  class GA\_Optimizer:  def \_\_init\_\_(self, param\_ranges, svm\_cap, svm\_life, scaler\_X, scaler\_cap, scaler\_life,  pop\_size=50, generations=100, base\_params=None):  self.param\_ranges = param\_ranges # 参数范围  self.n\_params = param\_ranges.shape[0]  self.svm\_cap = svm\_cap # 容量预测模型  self.svm\_life = svm\_life # 寿命预测模型  self.scaler\_X = scaler\_X  self.scaler\_cap = scaler\_cap  self.scaler\_life = scaler\_life  self.pop\_size = pop\_size  self.generations = generations  self.base\_params = base\_params if base\_params is not None else np.mean(param\_ranges, axis=1)    # 记录优化过程  self.history = []    # 初始化种群  def init\_population(self):  pop = []  for \_ in range(self.pop\_size):  individual = []  for i in range(self.n\_params):  # 在参数范围内随机生成  individual.append(random.uniform(  self.param\_ranges[i, 0], self.param\_ranges[i, 1]))  pop.append(np.array(individual))  return pop    # 适应度函数（多目标优化）  def fitness(self, individual):  # 转换为模型输入格式  individual\_scaled = self.scaler\_X.transform(individual.reshape(1, -1))    # 预测容量和寿命（反归一化）  cap\_pred\_scaled = self.svm\_cap.predict(individual\_scaled)[0]  cap\_pred = self.scaler\_cap.inverse\_transform([[cap\_pred\_scaled]])[0][0]    life\_pred\_scaled = self.svm\_life.predict(individual\_scaled)[0]  life\_pred = self.scaler\_life.inverse\_transform([[life\_pred\_scaled]])[0][0]    # 计算调整成本  cost = 200 \* np.abs(individual[0] - self.base\_params[0]) + \  1500 \* np.abs(individual[1] - self.base\_params[1]) + \  100 \* np.abs(individual[2] - self.base\_params[2]) + \  50 \* np.abs(individual[3] - self.base\_params[3]) + \  30 \* np.abs(individual[4] - self.base\_params[4])    # 约束惩罚  penalty = 0  if cap\_pred < 3.6: # 容量约束  penalty += (3.6 - cap\_pred) \* 10000  if life\_pred < 1000: # 寿命约束  penalty += (1000 - life\_pred) \* 10    # 多目标适应度计算（最大化容量和寿命，最小化成本）  # 权重：容量(40%)、寿命(40%)、成本(20%)  fitness\_score = (cap\_pred / 3.8) \* 40 + (life\_pred / 1200) \* 40 - (cost / 2000) \* 20  return fitness\_score - penalty # 减去惩罚    # 选择操作（锦标赛选择）  def select(self, pop, fitnesses, k=3):  selected = []  for \_ in range(2):  candidates = random.sample(list(zip(pop, fitnesses)), k)  candidates.sort(key=lambda x: x[1], reverse=True)  selected.append(candidates[0][0])  return selected    # 交叉操作  def crossover(self, p1, p2):  # 模拟二进制交叉  c1, c2 = p1.copy(), p2.copy()  for i in range(self.n\_params):  if random.random() < 0.5:  c1[i], c2[i] = c2[i], c1[i]  return c1, c2    # 变异操作  def mutate(self, individual, rate=0.1):  mutated = individual.copy()  for i in range(self.n\_params):  if random.random() < rate:  # 高斯变异  mutated[i] += random.gauss(0, 0.1 \* (self.param\_ranges[i, 1] - self.param\_ranges[i, 0]))  # 边界处理  mutated[i] = np.clip(mutated[i], self.param\_ranges[i, 0], self.param\_ranges[i, 1])  return mutated    # 优化主函数  def optimize(self):  pop = self.init\_population()  best\_individual = pop[0]  best\_fitness = self.fitness(best\_individual)    for gen in range(self.generations):  # 计算适应度  fitnesses = [self.fitness(ind) for ind in pop]    # 更新最优解  current\_best\_idx = np.argmax(fitnesses)  if fitnesses[current\_best\_idx] > best\_fitness:  best\_individual = pop[current\_best\_idx].copy()  best\_fitness = fitnesses[current\_best\_idx]    self.history.append(best\_fitness)    # 打印进度  if gen % 10 == 0:  print(f"迭代 {gen}, 最优适应度: {best\_fitness:.4f}")    # 生成下一代  new\_pop = [best\_individual] # 精英保留    while len(new\_pop) < self.pop\_size:  # 选择父母  p1, p2 = self.select(pop, fitnesses)  # 交叉  c1, c2 = self.crossover(p1, p2)  # 变异  c1 = self.mutate(c1)  c2 = self.mutate(c2)  # 添加到新种群  new\_pop.append(c1)  if len(new\_pop) < self.pop\_size:  new\_pop.append(c2)    pop = new\_pop    return best\_individual, best\_fitness  # 评估最优参数  def evaluate\_params(params, svm\_cap, svm\_life, scaler\_X, scaler\_cap, scaler\_life, base\_params):  # 预测性能  params\_scaled = scaler\_X.transform(params.reshape(1, -1))  cap\_pred = scaler\_cap.inverse\_transform([[svm\_cap.predict(params\_scaled)[0]]])[0][0]  life\_pred = scaler\_life.inverse\_transform([[svm\_life.predict(params\_scaled)[0]]])[0][0]    # 计算成本  cost = 200 \* np.abs(params[0] - base\_params[0]) + \  1500 \* np.abs(params[1] - base\_params[1]) + \  100 \* np.abs(params[2] - base\_params[2]) + \  50 \* np.abs(params[3] - base\_params[3]) + \  30 \* np.abs(params[4] - base\_params[4])    # 与原始经验参数对比  base\_scaled = scaler\_X.transform(base\_params.reshape(1, -1))  base\_cap = scaler\_cap.inverse\_transform([[svm\_cap.predict(base\_scaled)[0]]])[0][0]  base\_life = scaler\_life.inverse\_transform([[svm\_life.predict(base\_scaled)[0]]])[0][0]    print("\n===== 最优参数评估 =====")  print(f"参数组合:")  print(f" 固含量: {params[0]:.2f}% (基准: {base\_params[0]}%)")  print(f" 涂布速度: {params[1]:.2f}m/min (基准: {base\_params[1]}m/min)")  print(f" 辊压压力: {params[2]:.2f}MPa (基准: {base\_params[2]}MPa)")  print(f" 温度: {params[3]:.2f}℃ (基准: {base\_params[3]}℃)")  print(f" 时间: {params[4]:.2f}min (基准: {base\_params[4]}min)")  print(f"\n性能预测:")  print(f" 电芯容量: {cap\_pred:.3f}Ah (基准: {base\_cap:.3f}Ah, 提升: {(cap\_pred-base\_cap)/base\_cap\*100:.2f}%)")  print(f" 循环寿命: {life\_pred:.1f}次 (基准: {base\_life:.1f}次, 提升: {(life\_pred-base\_life)/base\_life\*100:.2f}%)")  print(f" 参数调整成本: {cost:.2f}元/批次")  print(f" 是否满足容量要求(≥3.6Ah): {'是' if cap\_pred >= 3.6 else '否'}")  print(f" 是否满足寿命要求(≥1000次): {'是' if life\_pred >= 1000 else '否'}")    return {  "capacity": cap\_pred,  "cycle\_life": life\_pred,  "cost": cost,  "base\_capacity": base\_cap,  "base\_life": base\_life  }  # 可视化结果  def visualize\_results(X, capacity, cycle\_life, best\_params):  fig, axes = plt.subplots(2, 2, figsize=(14, 12))    # 1. 固含量对容量的影响  axes[0,0].scatter(X[:,0], capacity, alpha=0.5, label='原始数据')  axes[0,0].scatter(best\_params[0], evaluate\_params(best\_params, svm\_cap, svm\_life, scaler\_X, scaler\_cap, scaler\_life, base\_params)['capacity'],  c='red', s=100, marker='\*', label='最优参数')  axes[0,0].axvline(x=base\_params[0], color='green', linestyle='--', label='基准参数')  axes[0,0].axhline(y=3.6, color='orange', linestyle='--', label='容量目标')  axes[0,0].set\_xlabel('固含量 (%)')  axes[0,0].set\_ylabel('电芯容量 (Ah)')  axes[0,0].set\_title('固含量对电芯容量的影响')  axes[0,0].legend()    # 2. 涂布速度对寿命的影响  axes[0,1].scatter(X[:,1], cycle\_life, alpha=0.5, label='原始数据')  axes[0,1].scatter(best\_params[1], evaluate\_params(best\_params, svm\_cap, svm\_life, scaler\_X, scaler\_cap, scaler\_life, base\_params)['cycle\_life'],  c='red', s=100, marker='\*', label='最优参数')  axes[0,1].axvline(x=base\_params[1], color='green', linestyle='--', label='基准参数')  axes[0,1].axhline(y=1000, color='orange', linestyle='--', label='寿命目标')  axes[0,1].set\_xlabel('涂布速度 (m/min)')  axes[0,1].set\_ylabel('循环寿命 (次)')  axes[0,1].set\_title('涂布速度对循环寿命的影响')  axes[0,1].legend()    # 3. 辊压压力对容量的影响  axes[1,0].scatter(X[:,2], capacity, alpha=0.5, label='原始数据')  axes[1,0].scatter(best\_params[2], evaluate\_params(best\_params, svm\_cap, svm\_life, scaler\_X, scaler\_cap, scaler\_life, base\_params)['capacity'],  c='red', s=100, marker='\*', label='最优参数')  axes[1,0].axvline(x=base\_params[2], color='green', linestyle='--', label='基准参数')  axes[1,0].set\_xlabel('辊压压力 (MPa)')  axes[1,0].set\_ylabel('电芯容量 (Ah)')  axes[1,0].set\_title('辊压压力对电芯容量的影响')  axes[1,0].legend()    # 4. 适应度进化曲线  axes[1,1].plot(ga\_optimizer.history)  axes[1,1].set\_xlabel('迭代次数')  axes[1,1].set\_ylabel('最优适应度')  axes[1,1].set\_title('遗传算法适应度进化曲线')  axes[1,1].grid(True)    plt.tight\_layout()  plt.show()  # 主函数  if \_\_name\_\_ == "\_\_main\_\_":  start\_time = time.time()    # 训练SVM预测模型  print("训练SVM预测模型...")  svm\_cap, svm\_life = train\_svm\_models()    # 初始化遗传算法优化器  ga\_optimizer = GA\_Optimizer(  param\_ranges=param\_ranges,  svm\_cap=svm\_cap,  svm\_life=svm\_life,  scaler\_X=scaler\_X,  scaler\_cap=scaler\_cap,  scaler\_life=scaler\_life,  pop\_size=50,  generations=100,  base\_params=base\_params  )    # 运行优化  print("\n开始遗传算法优化生产参数...")  best\_params, best\_fitness = ga\_optimizer.optimize()    # 评估最优参数  stats = evaluate\_params(best\_params, svm\_cap, svm\_life, scaler\_X, scaler\_cap, scaler\_life, base\_params)    # 可视化结果  visualize\_results(X, capacity, cycle\_life, best\_params)    print(f"\n总优化时间: {time.time() - start\_time:.2f}秒") |

### 5. 模糊逻辑（FL）+ 粒子群优化（PSO）组合模型案例题目

**题目：城市交叉口交通信号动态配时优化问题**

* **问题背景**：某城市主干道与次干道交叉口的早晚高峰拥堵严重，早高峰（7:30-9:00）平均等待时长达 15 分钟，晚高峰（17:30-19:00）因行人、非机动车混行导致通行效率下降 40%。当前信号配时为固定方案（如东西向绿灯 40 秒、南北向 30 秒），无法适应动态交通流。
* **问题描述**：需设计动态配时模型，根据实时交通流调整绿灯时长，目标包括：① 最小化车辆平均等待时间；② 最小化行人过街等待时间（≤60 秒）；③ 避免同一方向连续红灯超过 90 秒。输入信息含模糊性（如 “车流量较大”“行人较多”），需处理非机动车干扰等不确定性因素。
* **数据情况**：提供连续 7 天的交通流数据，包括：各方向每 5 分钟的车流量（辆）、行人过街数量（人）、非机动车流量（辆），以及不同配时方案下的实际等待时间记录。数据中含部分缺失值（如雨天传感器故障），需通过模糊处理方法兼容。

### 5. 模糊逻辑（FL）+ 粒子群优化（PSO）求解城市交叉口交通信号配时优化代码

|  |
| --- |
| import numpy as np  import random  import matplotlib.pyplot as plt  from sklearn.model\_selection import train\_test\_split  # 数据初始化与预处理  np.random.seed(42)  # 生成7天的交通流数据（每5分钟一条记录）  n\_samples = 7 \* 24 \* 12 # 7天×24小时×12个5分钟  # 输入特征：东西向车流量、南北向车流量、东西向行人数、南北向行人数  X = np.random.randint(5, 100, size=(n\_samples, 4)) # 流量范围5-100  # 输出标签：东西向绿灯时长、南北向绿灯时长（20-60秒）  y = np.zeros((n\_samples, 2))  for i in range(n\_samples):  # 简单规则生成标签（实际应用中可用历史数据）  y[i, 0] = 20 + X[i, 0] \* 0.4 + X[i, 2] \* 0.2 # 东西向绿灯时长  y[i, 1] = 20 + X[i, 1] \* 0.4 + X[i, 3] \* 0.2 # 南北向绿灯时长  y = np.clip(y, 20, 60) # 限制在20-60秒  # 划分训练集和测试集  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2)  # 模糊逻辑模块  class FuzzyLogic:  def \_\_init\_\_(self, n\_inputs=4, n\_outputs=2):  self.n\_inputs = n\_inputs  self.n\_outputs = n\_outputs  # 初始化隶属度函数参数（三角形：中心值和半宽度）  # 输入隶属度：少、中、多  self.input\_centers = np.array([[20, 50, 80]] \* n\_inputs) # 每个输入的3个中心值  self.input\_widths = np.array([[15, 15, 15]] \* n\_inputs) # 每个输入的3个半宽度  # 输出隶属度：短、中、长  self.output\_centers = np.array([[30, 40, 50]] \* n\_outputs) # 每个输出的3个中心值  # 三角形隶属度函数  def membership(self, x, centers, widths):  mu = np.zeros((len(x), len(centers)))  for i in range(len(centers)):  mu[:, i] = np.maximum(0, 1 - np.abs(x - centers[i]) / widths[i])  return mu  # 模糊推理  def infer(self, X):  n\_samples = X.shape[0]  outputs = np.zeros((n\_samples, self.n\_outputs))    for s in range(n\_samples):  # 计算输入隶属度  input\_mu = []  for i in range(self.n\_inputs):  mu = self.membership(X[s, i], self.input\_centers[i], self.input\_widths[i])  input\_mu.append(mu) # 每个输入的3个隶属度    # 组合所有规则（3^4=81条规则）  rule\_strengths = []  rule\_outputs = []  for a in range(3): # 输入1的隶属度索引  for b in range(3): # 输入2的隶属度索引  for c in range(3): # 输入3的隶属度索引  for d in range(3): # 输入4的隶属度索引  # 规则强度：取输入隶属度的最小值  strength = min(input\_mu[0][a], input\_mu[1][b],  input\_mu[2][c], input\_mu[3][d])  rule\_strengths.append(strength)  # 规则输出：取输出中心值（简单规则：输入隶属度索引平均值）  out = []  for o in range(self.n\_outputs):  out\_idx = int((a + b + c + d) / 4) # 简化规则  out.append(self.output\_centers[o, out\_idx])  rule\_outputs.append(out)    # 加权平均计算最终输出  total\_strength = sum(rule\_strengths)  if total\_strength == 0:  outputs[s] = np.mean(self.output\_centers, axis=1) # 默认值  else:  for o in range(self.n\_outputs):  weighted\_sum = sum([rs \* ro[o] for rs, ro in zip(rule\_strengths, rule\_outputs)])  outputs[s, o] = weighted\_sum / total\_strength    return np.clip(outputs, 20, 60) # 确保在有效范围内  # 将参数转换为粒子（用于PSO优化）  def params\_to\_particle(self):  particle = []  # 输入中心值  particle.extend(self.input\_centers.flatten())  # 输入半宽度  particle.extend(self.input\_widths.flatten())  # 输出中心值  particle.extend(self.output\_centers.flatten())  return np.array(particle)  # 从粒子更新参数  def particle\_to\_params(self, particle):  ptr = 0  # 输入中心值  self.input\_centers = particle[ptr:ptr+self.n\_inputs\*3].reshape(self.n\_inputs, 3)  ptr += self.n\_inputs\*3  # 输入半宽度  self.input\_widths = particle[ptr:ptr+self.n\_inputs\*3].reshape(self.n\_inputs, 3)  ptr += self.n\_inputs\*3  # 输出中心值  self.output\_centers = particle[ptr:ptr+self.n\_outputs\*3].reshape(self.n\_outputs, 3)  # PSO优化模块  class PSOOptimizer:  def \_\_init\_\_(self, fuzzy\_system, X\_train, y\_train, n\_particles=30, max\_iter=50):  self.fuzzy = fuzzy\_system  self.X = X\_train  self.y = y\_train  self.n\_particles = n\_particles  self.max\_iter = max\_iter  # 粒子维度：输入中心(4×3) + 输入宽度(4×3) + 输出中心(2×3) = 4×3×2 + 2×3 = 30  self.dim = 4\*3 + 4\*3 + 2\*3    # 初始化粒子和速度  self.global\_best\_particle = None  self.global\_best\_score = float('inf')  self.initialize\_particles()  # 初始化粒子  def initialize\_particles(self):  # 参数范围  input\_center\_min, input\_center\_max = 10, 90  input\_width\_min, input\_width\_max = 5, 25  output\_center\_min, output\_center\_max = 20, 60    self.particles = []  self.velocities = []  self.pbest\_particles = []  self.pbest\_scores = []    for \_ in range(self.n\_particles):  # 随机初始化粒子  particle = []  # 输入中心值  particle.extend(np.random.uniform(input\_center\_min, input\_center\_max, 4\*3))  # 输入半宽度  particle.extend(np.random.uniform(input\_width\_min, input\_width\_max, 4\*3))  # 输出中心值  particle.extend(np.random.uniform(output\_center\_min, output\_center\_max, 2\*3))  particle = np.array(particle)    # 初始化速度  velocity = np.random.uniform(-1, 1, self.dim)    # 计算初始适应度  self.fuzzy.particle\_to\_params(particle)  y\_pred = self.fuzzy.infer(self.X)  score = self.fitness\_score(y\_pred, self.y)    self.particles.append(particle)  self.velocities.append(velocity)  self.pbest\_particles.append(particle.copy())  self.pbest\_scores.append(score)    # 更新全局最优  if score < self.global\_best\_score:  self.global\_best\_score = score  self.global\_best\_particle = particle.copy()  # 适应度函数（最小化预测误差和等待时间）  def fitness\_score(self, y\_pred, y\_true):  # 预测误差（MSE）  mse = np.mean((y\_pred - y\_true)\*\*2)    # 惩罚项：行人等待时间过长（>60秒）  pedestrian\_wait = np.maximum(0, (y\_pred[:,0] + y\_pred[:,1]) - 60)  ped\_penalty = np.mean(pedestrian\_wait) \* 5    # 惩罚项：同一方向连续红灯过长（>90秒）  red\_light = 90 - y\_pred # 假设周期为90秒  red\_penalty = np.mean(np.maximum(0, red\_light)) \* 3    return mse + ped\_penalty + red\_penalty  # 执行PSO优化  def optimize(self):  w = 0.8 # 惯性权重  c1, c2 = 0.5, 0.5 # 学习因子    for iter in range(self.max\_iter):  for i in range(self.n\_particles):  # 更新速度  r1, r2 = np.random.rand(self.dim), np.random.rand(self.dim)  cognitive = c1 \* r1 \* (self.pbest\_particles[i] - self.particles[i])  social = c2 \* r2 \* (self.global\_best\_particle - self.particles[i])  self.velocities[i] = w \* self.velocities[i] + cognitive + social    # 更新位置  self.particles[i] += self.velocities[i]    # 约束处理（确保参数在合理范围）  self.particles[i][:12] = np.clip(self.particles[i][:12], 10, 90) # 输入中心  self.particles[i][12:24] = np.clip(self.particles[i][12:24], 5, 25) # 输入宽度  self.particles[i][24:] = np.clip(self.particles[i][24:], 20, 60) # 输出中心    # 计算适应度  self.fuzzy.particle\_to\_params(self.particles[i])  y\_pred = self.fuzzy.infer(self.X)  score = self.fitness\_score(y\_pred, self.y)    # 更新个体最优  if score < self.pbest\_scores[i]:  self.pbest\_particles[i] = self.particles[i].copy()  self.pbest\_scores[i] = score    # 更新全局最优  if score < self.global\_best\_score:  self.global\_best\_score = score  self.global\_best\_particle = self.particles[i].copy()    if iter % 5 == 0:  print(f"迭代 {iter}, 最优适应度: {self.global\_best\_score:.4f}")    # 应用最优参数  self.fuzzy.particle\_to\_params(self.global\_best\_particle)  return self.fuzzy  # 主函数  if \_\_name\_\_ == "\_\_main\_\_":  # 初始化模糊逻辑系统  fuzzy = FuzzyLogic()    # 用PSO优化模糊系统参数  pso = PSOOptimizer(fuzzy, X\_train, y\_train)  optimized\_fuzzy = pso.optimize()    # 测试优化效果  y\_pred = optimized\_fuzzy.infer(X\_test)  test\_mse = np.mean((y\_pred - y\_test)\*\* 2)  print(f"测试集MSE: {test\_mse:.4f}")    # 对比优化前后的结果（随机选10个样本）  sample\_idx = np.random.choice(len(X\_test), 10)  X\_sample = X\_test[sample\_idx]  y\_true\_sample = y\_test[sample\_idx]    # 优化前的模糊系统  original\_fuzzy = FuzzyLogic()  y\_pred\_original = original\_fuzzy.infer(X\_sample)  # 优化后的模糊系统  y\_pred\_optimized = optimized\_fuzzy.infer(X\_sample)    # 可视化对比  plt.figure(figsize=(12, 6))  x\_ticks = np.arange(10)    # 东西向绿灯时长对比  plt.subplot(1, 2, 1)  plt.bar(x\_ticks - 0.2, y\_true\_sample[:,0], 0.4, label='实际值')  plt.bar(x\_ticks + 0.2, y\_pred\_original[:,0], 0.4, label='优化前预测')  plt.bar(x\_ticks, y\_pred\_optimized[:,0], 0.2, label='优化后预测', color='green')  plt.title('东西向绿灯时长对比')  plt.legend()    # 南北向绿灯时长对比  plt.subplot(1, 2, 2)  plt.bar(x\_ticks - 0.2, y\_true\_sample[:,1], 0.4, label='实际值')  plt.bar(x\_ticks + 0.2, y\_pred\_original[:,1], 0.4, label='优化前预测')  plt.bar(x\_ticks, y\_pred\_optimized[:,1], 0.2, label='优化后预测', color='green')  plt.title('南北向绿灯时长对比')  plt.legend()    plt.tight\_layout()  plt.show() |

### 6. 深度学习（DNN）+ 梯度下降（GD）组合模型案例题目

**题目：区域电网短期负荷预测与发电调度优化问题**

* **问题背景**：某省级电网覆盖 10 个地级市，夏季用电高峰时需平衡火电机组、风电场、光伏电站的出力，避免弃风弃光（当前弃风率约 8%）或用电缺口。传统负荷预测误差达 10%，导致调度方案频繁调整，增加煤耗成本约 500 万元 / 月。
* **问题描述**：需先预测未来 24 小时的区域总用电负荷，再优化各电源的出力计划，目标包括：① 最小化预测误差（≤5%）；② 最小化发电总成本（火电煤耗 + 新能源补贴）；③ 确保实时出力与负荷的偏差≤2%。需考虑温度、节假日、工业停产等多因素对负荷的影响。
* **数据情况**：提供近 5 年的历史数据，包括：每小时的总用电负荷（MW）、平均温度（℃）、湿度（%）、风速（m/s）、日照时长（h）、节假日标识、工业用电量占比，以及各电源的单位发电成本、最大出力限制。

### 6. 深度学习（DNN）+ 梯度下降（GD）求解区域电网短期负荷预测与发电调度优化代码

|  |
| --- |
| import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  from sklearn.preprocessing import MinMaxScaler  from sklearn.metrics import mean\_absolute\_error, mean\_absolute\_percentage\_error  import tensorflow as tf  from tensorflow.keras.models import Sequential  from tensorflow.keras.layers import Dense, Dropout, BatchNormalization  from tensorflow.keras.optimizers import Adam  # 设置随机种子，保证结果可复现  np.random.seed(42)  tf.random.set\_seed(42)  # 数据生成与预处理  def generate\_power\_data(n\_years=5):  """生成包含多种影响因素的电网负荷数据"""  n\_days = n\_years \* 365  n\_hours = 24  n\_samples = n\_days \* n\_hours    # 创建时间特征  data = pd.DataFrame()  data['hour'] = np.tile(np.arange(n\_hours), n\_days) # 小时（0-23）  data['weekday'] = np.repeat(np.arange(7), n\_hours)[:n\_samples] # 星期（0-6）  data['month'] = np.repeat(np.tile(np.arange(1,13), [30\*n\_hours]), n\_days//12 +1)[:n\_samples] # 月份（1-12）    # 生成气象特征  data['temp'] = 15 + 12\*np.sin(np.linspace(0, 2\*np.pi\*n\_years, n\_samples)) + np.random.normal(0, 3, n\_samples) # 温度（℃）  data['humidity'] = 60 + 20\*np.sin(np.linspace(0, 2\*np.pi\*n\_years, n\_samples) + 1) + np.random.normal(0, 5, n\_samples) # 湿度（%）  data['wind\_speed'] = 3 + 2\*np.random.randn(n\_samples) # 风速（m/s）  data['sunshine'] = np.maximum(0, 8 + 4\*np.sin(np.linspace(0, 2\*np.pi\*n\_years, n\_samples) - 1) + np.random.normal(0, 1, n\_samples)) # 日照时长（h）    # 生成社会经济特征  data['holiday'] = np.zeros(n\_samples) # 节假日标识（0-1）  for d in range(0, n\_samples, n\_hours):  if np.random.rand() < 0.05: # 5%概率为节假日  data.loc[d:d+n\_hours-1, 'holiday'] = 1  data['industry\_ratio'] = 40 + 10\*np.random.randn(n\_samples) # 工业用电占比（%）  data['industry\_ratio'] = np.clip(data['industry\_ratio'], 20, 60)    # 生成目标负荷（MW）：受多种因素影响的非线性关系  base\_load = 1000 + 500\*np.sin(np.linspace(0, 2\*np.pi\*n\_years, n\_samples) + np.pi) # 基础负荷  hour\_effect = 200\*np.sin(data['hour']/24\*2\*np.pi + np.pi) # 小时效应  temp\_effect = -10 \* (data['temp'] - 15) # 温度效应  holiday\_effect = -200 \* data['holiday'] # 节假日效应  industry\_effect = 10 \* (data['industry\_ratio'] - 40) # 工业用电效应  noise = np.random.normal(0, 50, n\_samples) # 随机噪声    data['load'] = base\_load + hour\_effect + temp\_effect + holiday\_effect + industry\_effect + noise  data['load'] = np.clip(data['load'], 500, 2000) # 负荷范围限制    return data  # 生成5年数据  data = generate\_power\_data(n\_years=5)  print(f"生成数据规模: {data.shape}")  print(f"负荷数据范围: {data['load'].min():.2f} - {data['load'].max():.2f} MW")  # 特征与目标分离  features = ['hour', 'weekday', 'month', 'temp', 'humidity', 'wind\_speed', 'sunshine', 'holiday', 'industry\_ratio']  X = data[features].values  y = data['load'].values.reshape(-1, 1)  # 数据归一化  scaler\_X = MinMaxScaler()  scaler\_y = MinMaxScaler()  X\_scaled = scaler\_X.fit\_transform(X)  y\_scaled = scaler\_y.fit\_transform(y)  # 划分训练集与测试集（前4年训练，后1年测试）  train\_size = 4\*365\*24  X\_train, X\_test = X\_scaled[:train\_size], X\_scaled[train\_size:]  y\_train, y\_test = y\_scaled[:train\_size], y\_scaled[train\_size:]  # 构建DNN模型  def build\_dnn\_model(input\_dim):  """构建用于负荷预测的深度神经网络模型"""  model = Sequential([  Dense(128, activation='relu', input\_shape=(input\_dim,)),  BatchNormalization(),  Dropout(0.3),  Dense(64, activation='relu'),  BatchNormalization(),  Dropout(0.2),  Dense(32, activation='relu'),  Dense(1) # 输出层（负荷预测值）  ])    # 使用Adam优化器（基于梯度下降改进）  optimizer = Adam(learning\_rate=0.001, beta\_1=0.9, beta\_2=0.999)  model.compile(optimizer=optimizer, loss='mse')  return model  # 初始化并训练模型  model = build\_dnn\_model(input\_dim=X\_train.shape[1])  model.summary()  # 训练模型  history = model.fit(  X\_train, y\_train,  epochs=80,  batch\_size=64,  validation\_split=0.1,  verbose=1,  callbacks=[tf.keras.callbacks.EarlyStopping(patience=10, restore\_best\_weights=True)]  )  # 模型评估  def evaluate\_model(model, X\_test, y\_test, scaler\_y):  """评估模型预测性能"""  y\_pred\_scaled = model.predict(X\_test, verbose=0)  y\_pred = scaler\_y.inverse\_transform(y\_pred\_scaled)  y\_true = scaler\_y.inverse\_transform(y\_test)    mae = mean\_absolute\_error(y\_true, y\_pred)  mape = mean\_absolute\_percentage\_error(y\_true, y\_pred) \* 100    print(f"\n模型评估结果:")  print(f"平均绝对误差（MAE）: {mae:.2f} MW")  print(f"平均绝对百分比误差（MAPE）: {mape:.2f}%")    return y\_pred, y\_true  # 评估测试集性能  y\_pred, y\_true = evaluate\_model(model, X\_test, y\_test, scaler\_y)  # 发电调度优化模型  def power\_dispatch\_optimization(load\_pred, fossil\_max=1000, wind\_max=500, solar\_max=300, storage\_cap=200):  """  基于预测负荷进行发电调度优化  目标：最小化发电成本，最大化清洁能源利用率  """  n\_hours = len(load\_pred)    # 发电成本参数（元/MW）  costs = {  'fossil': 500, # 火电机组  'wind': 300, # 风电场  'solar': 200, # 光伏电站  'storage': 100 # 储能调用  }    # 初始化各电源出力  fossil = np.zeros(n\_hours)  wind = np.zeros(n\_hours)  solar = np.zeros(n\_hours)  storage = np.zeros(n\_hours)  storage\_level = np.zeros(n\_hours + 1) # 储能电量水平    # 模拟可再生能源出力（基于历史气象数据）  solar\_potential = np.maximum(0, 0.8 \* solar\_max \* np.sin(np.linspace(0, np.pi, 12)) +  0.2 \* solar\_max \* np.sin(np.linspace(0, np.pi, 12)))[np.tile(np.arange(12), 2)[:n\_hours]]  wind\_potential = wind\_max \* (0.5 + 0.5 \* np.random.rand(n\_hours))    # 逐小时调度  for t in range(n\_hours):  # 优先利用可再生能源  solar[t] = min(solar\_potential[t], load\_pred[t])  remaining = load\_pred[t] - solar[t]    wind[t] = min(wind\_potential[t], remaining)  remaining -= wind[t]    # 利用储能（优先放电）  if remaining > 0 and storage\_level[t] > 0:  use\_storage = min(remaining, storage\_level[t])  storage[t] = -use\_storage # 负号表示放电  storage\_level[t+1] = storage\_level[t] - use\_storage  remaining -= use\_storage    # 火电机组补充  fossil[t] = min(remaining, fossil\_max)  remaining -= fossil[t]    # 若仍有缺口，增加火电机组出力（允许10%过载）  if remaining > 0:  overload = min(remaining, 0.1 \* fossil\_max)  fossil[t] += overload  remaining -= overload    # 若有剩余电量，充电至储能  if remaining < 0:  charge = min(-remaining, storage\_cap - storage\_level[t])  storage[t] = charge # 正号表示充电  storage\_level[t+1] = storage\_level[t] + charge  else:  storage\_level[t+1] = storage\_level[t]    # 计算总成本  total\_cost = np.sum(fossil \* costs['fossil'] +  wind \* costs['wind'] +  solar \* costs['solar'] +  np.abs(storage) \* costs['storage'])    # 计算清洁能源占比  clean\_energy = np.sum(wind + solar + np.maximum(0, storage))  total\_energy = np.sum(fossil + wind + solar + np.abs(storage))  clean\_ratio = (clean\_energy / total\_energy) \* 100    print(f"\n调度优化结果:")  print(f"总调度成本: {total\_cost:.2f} 元")  print(f"清洁能源占比: {clean\_ratio:.2f}%")    return {  'fossil': fossil,  'wind': wind,  'solar': solar,  'storage': storage,  'total\_cost': total\_cost,  'clean\_ratio': clean\_ratio  }  # 对测试集最后30天进行调度优化  last\_month\_idx = -30\*24  load\_pred\_last\_month = y\_pred[last\_month\_idx:]  dispatch\_result = power\_dispatch\_optimization(load\_pred\_last\_month)  # 可视化结果  def plot\_results(y\_true, y\_pred, dispatch\_result, last\_month\_idx):  """可视化预测结果和调度方案"""  plt.figure(figsize=(18, 12))    # 1. 训练损失曲线  plt.subplot(3, 1, 1)  plt.plot(history.history['loss'], label='训练损失')  plt.plot(history.history['val\_loss'], label='验证损失')  plt.title('模型训练损失曲线')  plt.xlabel('Epoch')  plt.ylabel('均方误差（MSE）')  plt.legend()    # 2. 预测与实际负荷对比（最后7天）  last\_week\_idx = -7\*24  plt.subplot(3, 1, 2)  plt.plot(y\_true[last\_week\_idx:], label='实际负荷')  plt.plot(y\_pred[last\_week\_idx:], label='预测负荷', linestyle='--')  plt.title('最后7天负荷预测对比')  plt.xlabel('小时')  plt.ylabel('负荷（MW）')  plt.legend()    # 3. 发电调度方案（最后7天）  plt.subplot(3, 1, 3)  hours = np.arange(7\*24)  last\_7d\_idx = -7\*24  plt.stackplot(hours,  dispatch\_result['fossil'][last\_7d\_idx:],  dispatch\_result['wind'][last\_7d\_idx:],  dispatch\_result['solar'][last\_7d\_idx:],  np.abs(dispatch\_result['storage'][last\_7d\_idx:]),  labels=['火电', '风电', '光伏', '储能'])  plt.plot(load\_pred\_last\_month[last\_7d\_idx:], 'k--', label='预测负荷')  plt.title('最后7天发电调度方案')  plt.xlabel('小时')  plt.ylabel('出力（MW）')  plt.legend()    plt.tight\_layout()  plt.show()  # 可视化所有结果  plot\_results(y\_true, y\_pred, dispatch\_result, last\_month\_idx) |

### 7. 多目标遗传算法（MOGA）+ 层次分析法（AHP）组合模型案例题目

**题目：新能源汽车工厂选址与产能规划问题**

* **问题背景**：某车企计划在全国 5 个候选城市（如武汉、重庆、广州）中选择 2 个建设新能源汽车工厂，需考虑原材料运输成本、劳动力资源、政策补贴、环保要求等因素。不同城市的土地价格、供应链成熟度差异显著，且工厂投产后需确定各厂的年产能（5-20 万辆）。
* **问题描述**：需选择最优选址方案并分配产能，目标包括：① 最小化年均总成本（土地 + 运输 + 人力）；② 最大化政策补贴总额（部分城市对新能源项目补贴 20%-30%）；③ 最小化碳排放（≤50 万吨 / 年）。需结合专家意见（如供应链稳定性权重）进行多目标决策。
* **数据情况**：提供各候选城市的基础数据：土地价格（万元 / 亩）、劳动力成本（元 / 小时）、主要零部件供应商的距离及运输成本（元 / 辆）、当地环保排放标准、政策补贴细则，以及不同产能下的固定成本和变动成本核算表。

### 7. 多目标遗传算法（MOGA）+ 层次分析法（AHP）新能源汽车工厂选址与产能规划代码

|  |
| --- |
| import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  import random  from scipy.stats import rankdata  from matplotlib.patches import Patch  import seaborn as sns  # 设置随机种子，保证结果可复现  np.random.seed(42)  random.seed(42)  # 候选城市数据初始化  def initialize\_candidate\_cities():  """初始化10个候选城市的各项指标数据"""  cities = {  0: {'name': '武汉', 'land\_price': 85, 'labor\_cost': 28, 'transport': 1.2,  'subsidy': 0.25, 'carbon\_limit': 45, 'supply\_chain': 0.85, 'market': 0.82},  1: {'name': '重庆', 'land\_price': 65, 'labor\_cost': 22, 'transport': 1.4,  'subsidy': 0.3, 'carbon\_limit': 50, 'supply\_chain': 0.78, 'market': 0.75},  2: {'name': '广州', 'land\_price': 110, 'labor\_cost': 32, 'transport': 1.0,  'subsidy': 0.22, 'carbon\_limit': 42, 'supply\_chain': 0.90, 'market': 0.90},  3: {'name': '西安', 'land\_price': 55, 'labor\_cost': 20, 'transport': 1.6,  'subsidy': 0.35, 'carbon\_limit': 55, 'supply\_chain': 0.72, 'market': 0.68},  4: {'name': '上海', 'land\_price': 130, 'labor\_cost': 38, 'transport': 0.9,  'subsidy': 0.18, 'carbon\_limit': 38, 'supply\_chain': 0.95, 'market': 0.95},  5: {'name': '成都', 'land\_price': 70, 'labor\_cost': 25, 'transport': 1.3,  'subsidy': 0.28, 'carbon\_limit': 48, 'supply\_chain': 0.80, 'market': 0.78},  6: {'name': '沈阳', 'land\_price': 60, 'labor\_cost': 23, 'transport': 1.5,  'subsidy': 0.32, 'carbon\_limit': 52, 'supply\_chain': 0.70, 'market': 0.65},  7: {'name': '杭州', 'land\_price': 100, 'labor\_cost': 30, 'transport': 1.1,  'subsidy': 0.20, 'carbon\_limit': 40, 'supply\_chain': 0.88, 'market': 0.85},  8: {'name': '深圳', 'land\_price': 120, 'labor\_cost': 35, 'transport': 0.95,  'subsidy': 0.20, 'carbon\_limit': 35, 'supply\_chain': 0.92, 'market': 0.92},  9: {'name': '合肥', 'land\_price': 75, 'labor\_cost': 26, 'transport': 1.35,  'subsidy': 0.30, 'carbon\_limit': 47, 'supply\_chain': 0.83, 'market': 0.70}  }  return cities  # 层次分析法（AHP）计算权重  def ahp\_weight\_calculation():  """  基于专家判断矩阵计算各目标权重  目标包括：成本、政策补贴、碳排放、供应链完整性、市场潜力  """  # 构建判断矩阵（5x5）  # 行/列：0-成本，1-补贴，2-碳排放，3-供应链，4-市场  judge\_matrix = np.array([  [1, 3, 5, 2, 2], # 成本  [1/3, 1, 3, 1/2, 1/2], # 补贴  [1/5, 1/3, 1, 1/3, 1/3], # 碳排放  [1/2, 2, 3, 1, 1], # 供应链  [1/2, 2, 3, 1, 1] # 市场  ])    # 计算权重（特征值法）  eigvals, eigvecs = np.linalg.eig(judge\_matrix)  max\_idx = np.argmax(eigvals)  weights = eigvecs[:, max\_idx].real  weights = np.abs(weights) # 确保非负  weights /= np.sum(weights) # 归一化    print("AHP计算的各目标权重:")  print(f"成本权重: {weights[0]:.4f}")  print(f"补贴权重: {weights[1]:.4f}")  print(f"碳排放权重: {weights[2]:.4f}")  print(f"供应链权重: {weights[3]:.4f}")  print(f"市场权重: {weights[4]:.4f}")    return weights  # 多目标遗传算法参数设置  def set\_moga\_parameters():  """设置多目标遗传算法参数"""  return {  'pop\_size': 80, # 种群规模  'generations': 150, # 迭代次数  'mutation\_rate': 0.25, # 变异率  'n\_objectives': 5, # 目标函数数量  'n\_cities': 10, # 候选城市数量  'n\_plants': 2, # 计划建设工厂数量  'min\_capacity': 5, # 最小产能（万辆/年）  'max\_capacity': 25, # 最大产能（万辆/年）  'total\_demand': 35 # 总需求（万辆/年）  }  # 初始化种群  def initialize\_population(params):  """初始化遗传算法种群"""  pop = []  for \_ in range(params['pop\_size']):  # 随机选择n\_plants个不同的城市  selected\_cities = random.sample(range(params['n\_cities']), params['n\_plants'])    # 分配产能（总和接近总需求）  capacities = []  remaining = params['total\_demand']  for i in range(params['n\_plants'] - 1):  cap = random.uniform(params['min\_capacity'], min(params['max\_capacity'], remaining - params['min\_capacity']))  capacities.append(cap)  remaining -= cap  capacities.append(remaining) # 最后一个工厂承担剩余产能    # 确保产能在有效范围内  capacities = [np.clip(c, params['min\_capacity'], params['max\_capacity']) for c in capacities]    # 构建个体：(城市1, 产能1, 城市2, 产能2, ...)  individual = []  for i in range(params['n\_plants']):  individual.extend([selected\_cities[i], capacities[i]])    pop.append(tuple(individual))    return pop  # 目标函数计算  def calculate\_objectives(individual, cities, params):  """计算个体的各项目标函数值"""  n\_plants = params['n\_plants']  total\_cost = 0  total\_subsidy = 0  total\_carbon = 0  avg\_supply\_chain = 0  avg\_market = 0    # 工厂固定成本参数（亿元）  fixed\_cost\_coef = 2.5 # 每万辆产能的固定成本系数  var\_cost\_coef = 0.8 # 每万辆产能的可变成本系数    for i in range(n\_plants):  city\_idx = individual[2\*i]  capacity = individual[2\*i + 1]  city = cities[city\_idx]    # 1. 成本目标（土地+固定+可变+运输）  land\_cost = city['land\_price'] \* capacity \* 0.01 # 土地成本（亿元）  fixed\_cost = fixed\_cost\_coef \* capacity # 固定成本（亿元）  var\_cost = var\_cost\_coef \* capacity # 可变成本（亿元）  transport\_cost = city['transport'] \* capacity \* 0.1 # 运输成本（亿元）  total\_cost += land\_cost + fixed\_cost + var\_cost + transport\_cost    # 2. 补贴目标（投资额×补贴率）  investment = land\_cost + fixed\_cost  total\_subsidy += investment \* city['subsidy']    # 3. 碳排放目标（万辆对应碳排放量）  carbon\_emission = np.minimum(capacity \* 0.8, city['carbon\_limit']) # 假设每万辆排放0.8万吨  total\_carbon += carbon\_emission    # 4. 供应链完整性（平均值）  avg\_supply\_chain += city['supply\_chain']    # 5. 市场潜力（平均值）  avg\_market += city['market']    # 平均供应链和市场潜力  avg\_supply\_chain /= n\_plants  avg\_market /= n\_plants    # 返回目标值（成本、-补贴、碳排放、-供应链、-市场，统一为最小化问题）  return (  total\_cost,  -total\_subsidy,  total\_carbon,  -avg\_supply\_chain,  -avg\_market  )  # 非支配排序  def non\_dominated\_sorting(pop, cities, params):  """对种群进行非支配排序"""  n = len(pop)  objectives = [calculate\_objectives(ind, cities, params) for ind in pop]    # 初始化支配关系  dominated = [[] for \_ in range(n)] # 被该个体支配的个体列表  counts = np.zeros(n) # 支配该个体的个体数量  ranks = np.zeros(n) # 个体的排序等级    # 计算支配关系  for i in range(n):  for j in range(n):  if i != j:  # 检查i是否支配j  if all(o\_i <= o\_j for o\_i, o\_j in zip(objectives[i], objectives[j])):  dominated[i].append(j)  counts[j] += 1    # 分配排序等级  rank = 0  while np.any(counts == 0):  for i in range(n):  if counts[i] == 0:  ranks[i] = rank  counts[i] = -1 # 标记为已处理  # 减少被其支配的个体的计数  for j in dominated[i]:  counts[j] -= 1  rank += 1    return ranks  # 拥挤度计算  def crowding\_distance(objectives):  """计算种群中个体的拥挤度"""  n = len(objectives)  n\_obj = len(objectives[0])  dist = np.zeros(n)    for m in range(n\_obj):  # 按第m个目标排序  sorted\_indices = np.argsort([obj[m] for obj in objectives])  sorted\_objs = [objectives[i][m] for i in sorted\_indices]    # 边界个体的拥挤度设为无穷大  dist[sorted\_indices[0]] = np.inf  dist[sorted\_indices[-1]] = np.inf    # 计算中间个体的拥挤度  if sorted\_objs[-1] != sorted\_objs[0]: # 避免除以零  for i in range(1, n-1):  dist[sorted\_indices[i]] += (sorted\_objs[i+1] - sorted\_objs[i-1]) / (sorted\_objs[-1] - sorted\_objs[0])    return dist  # 选择操作（锦标赛选择）  def selection(pop, ranks, distances, params):  """锦标赛选择操作"""  selected = []  while len(selected) < params['pop\_size']:  # 随机选择两个个体  a, b = random.sample(range(len(pop)), 2)    # 选择排序等级低的个体，等级相同则选择拥挤度高的  if ranks[a] < ranks[b] or (ranks[a] == ranks[b] and distances[a] > distances[b]):  selected.append(pop[a])  else:  selected.append(pop[b])    return selected  # 交叉操作  def crossover(p1, p2, params):  """交叉操作生成子代个体"""  n\_plants = params['n\_plants']  # 随机选择交叉点  cross\_point = random.randint(1, 2\*n\_plants - 1)    # 生成子代  child1 = p1[:cross\_point] + p2[cross\_point:]  child2 = p2[:cross\_point] + p1[cross\_point:]    # 确保城市不重复  for child in [child1, child2]:  cities\_in\_child = [child[2\*i] for i in range(n\_plants)]  # 检查重复城市  if len(set(cities\_in\_child)) < n\_plants:  # 替换重复城市  all\_cities = set(range(params['n\_cities']))  used\_cities = set(cities\_in\_child)  available\_cities = list(all\_cities - used\_cities)    # 找出重复的位置  for i in range(n\_plants):  if cities\_in\_child.count(cities\_in\_child[i]) > 1:  # 替换为可用城市  if available\_cities:  child[2\*i] = available\_cities.pop(0)    return tuple(child1), tuple(child2)  # 变异操作  def mutate(individual, params, mutation\_rate=None):  """变异操作"""  if mutation\_rate is None:  mutation\_rate = params['mutation\_rate']    n\_plants = params['n\_plants']  mutated = list(individual)    # 城市变异  if random.random() < mutation\_rate:  plant\_idx = random.randint(0, n\_plants - 1)  current\_city = mutated[2\*plant\_idx]  # 选择不同的城市  other\_cities = [c for c in range(params['n\_cities']) if c != current\_city]  # 确保不与其他工厂城市重复  existing\_cities = [mutated[2\*i] for i in range(n\_plants) if i != plant\_idx]  possible\_cities = [c for c in other\_cities if c not in existing\_cities]  if possible\_cities:  mutated[2\*plant\_idx] = random.choice(possible\_cities)    # 产能变异  if random.random() < mutation\_rate:  plant\_idx = random.randint(0, n\_plants - 1)  # 小幅调整产能  current\_cap = mutated[2\*plant\_idx + 1]  change = random.uniform(-2, 2) # 产能变化范围  new\_cap = current\_cap + change  # 确保在有效范围内  new\_cap = np.clip(new\_cap, params['min\_capacity'], params['max\_capacity'])  mutated[2\*plant\_idx + 1] = new\_cap    # 调整其他工厂产能以保持总产能稳定  if n\_plants > 1:  other\_idx = random.choice([i for i in range(n\_plants) if i != plant\_idx])  mutated[2\*other\_idx + 1] -= change  mutated[2\*other\_idx + 1] = np.clip(  mutated[2\*other\_idx + 1],  params['min\_capacity'],  params['max\_capacity']  )    return tuple(mutated)  # 多目标遗传算法主函数  def moga\_optimization(cities, params, weights):  """执行多目标遗传算法优化"""  # 初始化种群  pop = initialize\_population(params)  best\_solutions = []  best\_scores = []    for gen in range(params['generations']):  # 计算非支配排序和拥挤度  ranks = non\_dominated\_sorting(pop, cities, params)  objectives = [calculate\_objectives(ind, cities, params) for ind in pop]  distances = crowding\_distance(objectives)    # 选择操作  selected = selection(pop, ranks, distances, params)    # 交叉操作  offspring = []  for i in range(0, params['pop\_size'], 2):  p1 = selected[i]  p2 = selected[i+1] if i+1 < params['pop\_size'] else selected[0]  c1,</doubaocanvas> |

### 8. 模拟退火（SA）+ 线性规划（LP）组合模型案例题目

**题目：突发公共卫生事件医疗物资分配优化问题**

* **问题背景**：某省突发疫情，15 个地市出现病例，需紧急分配口罩、防护服、呼吸机等医疗物资。现有省级储备库 3 个，物资总量有限（如口罩 500 万只、防护服 10 万套），部分地市因交通管制导致运输难度增加。
* **问题描述**：需制定物资分配方案，目标包括：① 最大化物资覆盖率（高风险地区优先）；② 最小化运输时间（高风险地区≤24 小时）；③ 最小化运输成本（与距离和物资重量相关）。分配需满足线性约束（如某物资总分配量≤库存量），同时处理交通管制等非线性因素。
* **数据情况**：提供各地市的风险等级（高 / 中 / 低）、确诊病例数、床位数、各物资需求量，储备库的物资库存量，地市与储备库的距离及运输成本系数（元 / 吨・km），以及受交通管制影响的路段运输时间修正系数（如增加 50%）。

### 8. 模拟退火（SA）+ 线性规划（LP）求解突发公共卫生事件医疗物资分配优化代码

|  |
| --- |
| import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  import random  from math import exp  from scipy.optimize import linprog  import seaborn as sns  from matplotlib.colors import LinearSegmentedColormap  # 设置随机种子，保证结果可复现  np.random.seed(42)  random.seed(42)  # 数据初始化  def initialize\_medical\_data():  """初始化医疗物资分配问题的基础数据"""  # 15个受灾地区，3个储备仓库，3类物资  n\_regions = 15  n\_warehouses = 3  n\_items = 3 # 0-防护装备，1-急救药品，2-医疗器械    # 各地区物资需求量（单位：防护装备-箱，药品-箱，器械-套）  demand = np.array([  [120, 80, 30], [90, 60, 20], [100, 70, 25],  [150, 90, 35], [80, 50, 15], [110, 65, 22],  [95, 68, 24], [130, 85, 32], [70, 45, 12],  [160, 95, 38], [92, 58, 21], [105, 72, 26],  [125, 82, 31], [85, 52, 16], [115, 75, 28]  ])    # 各仓库物资库存量  stock = np.array([  [600, 400, 150], # 仓库1  [500, 350, 130], # 仓库2  [450, 300, 120] # 仓库3  ])    # 地区风险等级（0-低，1-中，2-高）  risk\_level = np.array([2, 1, 1, 2, 0, 1, 1, 2, 0, 2, 1, 1, 2, 0, 1])    # 地区坐标（用于可视化）  region\_coords = np.random.rand(n\_regions, 2) \* 100  warehouse\_coords = np.array([[10, 10], [50, 80], [90, 30]]) # 仓库固定位置    # 运输成本系数（万元/单位物资·公里）  cost\_coef = np.array([0.08, 0.15, 0.25]) # 防护装备、急救药品、医疗器械    # 仓库到地区的距离（公里）  distance = np.zeros((n\_warehouses, n\_regions))  for i in range(n\_warehouses):  for j in range(n\_regions):  distance[i, j] = np.sqrt(np.sum((warehouse\_coords[i] - region\_coords[j])\*\*2))    # 通行时间修正系数（1为正常，>1表示通行困难）  time\_factor = np.ones((n\_warehouses, n\_regions))  # 随机设置20%的路段为困难路段  for i in range(n\_warehouses):  for j in range(n\_regions):  if random.random() < 0.2:  time\_factor[i, j] = 1.5 + random.random() \* 0.5 # 1.5-2.0倍时间    # 地区名称  region\_names = [f"地区{i+1}" for i in range(n\_regions)]  warehouse\_names = [f"仓库{i+1}" for i in range(n\_warehouses)]  item\_names = ["防护装备", "急救药品", "医疗器械"]    return {  "n\_regions": n\_regions,  "n\_warehouses": n\_warehouses,  "n\_items": n\_items,  "demand": demand,  "stock": stock,  "risk\_level": risk\_level,  "region\_coords": region\_coords,  "warehouse\_coords": warehouse\_coords,  "cost\_coef": cost\_coef,  "distance": distance,  "time\_factor": time\_factor,  "region\_names": region\_names,  "warehouse\_names": warehouse\_names,  "item\_names": item\_names  }  # 线性规划模型（固定分配方案下的物资优化）  def lp物资分配(data, warehouse, regions):  """  对指定仓库和负责地区进行物资分配优化  目标：最小化运输成本，满足需求和库存约束  """  n\_regions = len(regions)  n\_items = data["n\_items"]  n\_vars = n\_regions \* n\_items # 变量数量：地区×物资    # 目标函数系数（总运输成本）  c = []  for r\_idx, r in enumerate(regions):  for item in range(n\_items):  # 成本 = 距离×单位成本×时间系数  cost = data["distance"][warehouse, r] \* data["cost\_coef"][item] \* data["time\_factor"][warehouse, r]  c.append(cost)    # 约束条件：A\_ub @ x <= b\_ub  A\_ub = []  b\_ub = []    # 1. 仓库库存约束  for item in range(n\_items):  row = [0] \* n\_vars  for r\_idx in range(n\_regions):  row[r\_idx \* n\_items + item] = 1 # 对应物资的变量系数为1  A\_ub.append(row)  b\_ub.append(data["stock"][warehouse, item]) # 库存上限    # 2. 地区需求约束（按风险等级设置最低满足率）  for r\_idx, r in enumerate(regions):  risk = data["risk\_level"][r]  for item in range(n\_items):  row = [0] \* n\_vars  row[r\_idx \* n\_items + item] = -1 # 负号用于转换为 <= 约束  A\_ub.append(row)    # 高风险90%，中风险80%，低风险70%  min\_rate = 0.9 if risk == 2 else 0.8 if risk == 1 else 0.7  min\_demand = data["demand"][r, item] \* min\_rate  b\_ub.append(-min\_demand) # 负号对应不等式转换    # 求解线性规划  bounds = [(0, None) for \_ in range(n\_vars)] # 变量非负约束  result = linprog(c, A\_ub=A\_ub, b\_ub=b\_ub, bounds=bounds, method='highs')    if result.success:  # 重塑结果为（地区×物资）矩阵  allocation = result.x.reshape(n\_regions, n\_items)  return allocation, result.fun # 分配方案和总成本  else:  # 若无解，返回高成本方案（实际应用中需调整约束）  return np.zeros((n\_regions, n\_items)), 1e10  # 模拟退火算法（优化仓库-地区分配关系）  def sa\_optimization(data):  """  模拟退火算法优化仓库-地区分配方案  目标：找到全局最优的物资配送网络  """  n\_regions = data["n\_regions"]  n\_warehouses = data["n\_warehouses"]    # 初始解：随机分配地区给仓库  current\_assign = np.random.randint(0, n\_warehouses, size=n\_regions)  best\_assign = current\_assign.copy()    # 计算方案总成本  def total\_cost(assignment):  """计算当前分配方案的总成本"""  cost = 0  for w in range(n\_warehouses):  regions = np.where(assignment == w)[0]  if len(regions) == 0:  continue  \_, w\_cost = lp物资分配(data, w, regions)  cost += w\_cost  return cost    # 初始成本  current\_cost = total\_cost(current\_assign)  best\_cost = current\_cost  cost\_history = [best\_cost]    # 模拟退火参数  temp = 1000.0 # 初始温度  cooling\_rate = 0.95 # 冷却速率  iterations = 200 # 迭代次数  min\_temp = 1e-3 # 最低温度    print("模拟退火优化过程:")  print(f"初始成本: {current\_cost:.2f} 万元")    for i in range(iterations):  if temp < min\_temp:  break    # 生成邻域解（随机交换两个地区的分配）  a, b = random.sample(range(n\_regions), 2)  new\_assign = current\_assign.copy()  new\_assign[a], new\_assign[b] = new\_assign[b], new\_assign[a]  new\_cost = total\_cost(new\_assign)    # 接受准则  if new\_cost < current\_cost:  # 接受更优解  current\_assign = new\_assign  current\_cost = new\_cost  if new\_cost < best\_cost:  best\_cost = new\_cost  best\_assign = new\_assign  else:  # Metropolis准则接受较差解  acceptance\_prob = exp((current\_cost - new\_cost) / temp)  if random.random() < acceptance\_prob:  current\_assign = new\_assign  current\_cost = new\_cost    # 降温  temp \*= cooling\_rate  cost\_history.append(best\_cost)    # 定期输出进度  if (i + 1) % 20 == 0:  print(f"迭代 {i+1}, 温度: {temp:.4f}, 最优成本: {best\_cost:.2f} 万元")    print(f"优化完成, 最终最优成本: {best\_cost:.2f} 万元")  return best\_assign, best\_cost, cost\_history  # 生成详细分配结果  def generate\_allocation\_details(data, best\_assign):  """根据最优分配方案生成详细的物资分配表"""  n\_warehouses = data["n\_warehouses"]  allocation = np.zeros((n\_warehouses, data["n\_regions"], data["n\_items"]))  coverage = np.zeros((data["n\_regions"], data["n\_items"])) # 需求满足率    for w in range(n\_warehouses):  regions = np.where(best\_assign == w)[0]  if len(regions) == 0:  continue  alloc, \_ = lp物资分配(data, w, regions)  for r\_idx, r in enumerate(regions):  allocation[w, r] = alloc[r\_idx]  coverage[r] = alloc[r\_idx] / data["demand"][r] \* 100 # 转换为百分比    return allocation, coverage  # 结果可视化  def visualize\_results(data, best\_assign, allocation, coverage, cost\_history):  """可视化物资分配结果"""  plt.figure(figsize=(18, 14))    # 1. 成本变化曲线  plt.subplot(2, 2, 1)  plt.plot(cost\_history)  plt.title('模拟退火优化成本变化', fontsize=12)  plt.xlabel('迭代次数')  plt.ylabel('总成本（万元）')  plt.grid(True, linestyle='--', alpha=0.7)    # 2. 地区-仓库分配地图  plt.subplot(2, 2, 2)  cmap = LinearSegmentedColormap.from\_list('warehouse\_cmap', ['#FF9999', '#66B2FF', '#99FF99'])    # 绘制地区  scatter = plt.scatter(  data["region\_coords"][:, 0], data["region\_coords"][:, 1],  c=best\_assign, cmap=cmap, s=100, edgecolors='k', alpha=0.7  )  # 绘制仓库  plt.scatter(  data["warehouse\_coords"][:, 0], data["warehouse\_coords"][:, 1],  c=np.arange(data["n\_warehouses"]), cmap=cmap, s=300, marker='\*', edgecolors='k'  )  # 绘制分配关系  for r in range(data["n\_regions"]):  w = best\_assign[r]  plt.plot(  [data["region\_coords"][r, 0], data["warehouse\_coords"][w, 0]],  [data["region\_coords"][r, 1], data["warehouse\_coords"][w, 1]],  'gray', linestyle='--', alpha=0.3  )    plt.title('地区-仓库分配关系', fontsize=12)  plt.xlabel('X坐标')  plt.ylabel('Y坐标')  legend\_elements = [Patch(facecolor=cmap(i), edgecolor='k', label=f'仓库{i+1}')  for i in range(data["n\_warehouses"])]  plt.legend(handles=legend\_elements, loc='upper right')    # 3. 需求满足率热力图  plt.subplot(2, 1, 2)  region\_names = [f"地区{i+1}" for i in range(data["n\_regions"])]  sns.heatmap(coverage, annot=True, fmt='.1f', cmap='YlGn',  xticklabels=data["item\_names"], yticklabels=region\_names)  plt.title('各地区物资需求满足率（%）', fontsize=12)  plt.tight\_layout()    plt.show()    # 4. 各仓库物资分配量（条形图）  plt.figure(figsize=(15, 6))  bar\_width = 0.25  index = np.arange(data["n\_items"])    for w in range(data["n\_warehouses"]):  total\_alloc = np.sum(allocation[w], axis=0)  plt.bar(index + w \* bar\_width, total\_alloc, bar\_width,  label=data["warehouse\_names"][w])    plt.title('各仓库物资分配总量', fontsize=12)  plt.xlabel('物资类型')  plt.ylabel('分配数量')  plt.xticks(index + bar\_width, data["item\_names"])  plt.legend()  plt.grid(axis='y', linestyle='--', alpha=0.7)  plt.tight\_layout()  plt.show()  # 主函数  def main():  # 初始化数据  data = initialize\_medical\_data()    # 执行模拟退火优化  best\_assign, best\_cost, cost\_history = sa\_optimization(data)    # 生成详细分配结果  allocation, coverage = generate\_allocation\_details(data, best\_assign)    # 输出关键结果  print("\n优化结果摘要:")  for w in range(data["n\_warehouses"]):  regions = np.where(best\_assign == w)[0]  regions\_str = ", ".join([f"地区{r+1}" for r in regions])  print(f"仓库{w+1}负责的地区: {regions\_str}")    print(f"\n各风险等级平均满足率:")  for risk in [0, 1, 2]:  risk\_regions = np.where(data["risk\_level"] == risk)[0]  if len(risk\_regions) > 0:  avg\_cov = np.mean(coverage[risk\_regions])  print(f"{'低' if risk==0 else '中' if risk==1 else '高'}风险地区: {avg\_cov:.2f}%")    # 可视化结果  visualize\_results(data, best\_assign, allocation, coverage, cost\_history)  if \_\_name\_\_ == "\_\_main\_\_":  main() |

### 9. 贝叶斯优化（BO）+ 随机森林（RF）组合模型案例题目

**题目：新型光伏材料配方优化问题**

* **问题背景**：某材料实验室研发高效光伏组件，其核心材料的转化率受 5 种原料的配比影响（如硅片纯度、掺杂元素比例）。每次实验需 3 天且成本约 2 万元，传统试错法已进行 50 次实验，转化率最高仅 22%，远低于目标 25%。
* **问题描述**：需通过少量实验找到最优配方，目标是最大化光伏材料的光电转化率，约束条件为原料成本≤500 元 / 片，材料稳定性（耐温范围 - 40℃至 85℃）达标。需构建代理模型减少实验次数，兼顾优化效率与成本。
* **数据情况**：提供已完成的 50 次实验数据，包括 5 种原料的配比（如硅纯度 99.99%-99.999%）、实验环境参数（温度、湿度）、测得的转化率（%）、稳定性测试结果，以及各原料的单价。

### 9. 贝叶斯优化（BO）+ 随机森林（RF）求解新型光伏材料配方优化代码

|  |
| --- |
| import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  import seaborn as sns  from sklearn.ensemble import RandomForestRegressor  from sklearn.model\_selection import cross\_val\_score  from skopt import gp\_minimize  from skopt.space import Real  from skopt.plots import plot\_convergence  from skopt.utils import use\_named\_args  # 设置随机种子，保证结果可复现  np.random.seed(42)  # 数据生成（模拟光伏材料实验数据）  def generate\_material\_data(n\_initial=50):  """生成光伏材料配方与性能的模拟实验数据"""  # 配方参数范围：  # x1: 硅纯度（99.99%-99.999%）  # x2: 掺杂元素A比例（0.01%-0.1%）  # x3: 掺杂元素B比例（0.005%-0.05%）  # x4: 薄膜厚度（100-500nm）  # x5: 退火温度（500-1000℃）    # 生成初始实验参数  data = pd.DataFrame()  data['x1'] = np.random.uniform(99.99, 99.999, n\_initial)  data['x2'] = np.random.uniform(0.01, 0.1, n\_initial)  data['x3'] = np.random.uniform(0.005, 0.05, n\_initial)  data['x4'] = np.random.uniform(100, 500, n\_initial)  data['x5'] = np.random.uniform(500, 1000, n\_initial)    # 生成光电转化率（%）：基于非线性函数模拟  # 基础转化率（与硅纯度正相关）  base\_eff = 18 + 2 \* ((data['x1'] - 99.99) / 0.009)  # 掺杂元素影响（存在最优比例）  dopant\_eff = 3 \* np.exp(-5 \* (data['x2'] - 0.05)\*\*2) + 2 \* np.exp(-10 \* (data['x3'] - 0.02)\*\* 2)  # 薄膜厚度和退火温度影响  film\_temp\_eff = 1.5 \* np.sin(data['x4']/500 \* np.pi) + 2 \* np.sin(data['x5']/1000 \* np.pi)  # 随机测量误差  noise = np.random.normal(0, 0.3, n\_initial)    data['efficiency'] = base\_eff + dopant\_eff + film\_temp\_eff + noise  data['efficiency'] = np.clip(data['efficiency'], 17, 23) # 限制在合理范围内    # 生成制造成本（元/片）  data['cost'] = 100 + 50\*(data['x1'] - 99.99)/0.009 + 1000\*data['x2'] + 2000\*data['x3'] + 0.1\*data['x4'] + 0.05\*data['x5']    # 生成稳定性指标（-40℃至85℃循环测试后的衰减率）  data['stability'] = 1 - (0.02 \* (data['x4'] < 200) + 0.03 \* (data['x5'] < 600) + np.random.normal(0, 0.01, n\_initial))  data['stability'] = np.clip(data['stability'], 0.85, 0.99)    return data  # 定义参数空间  param\_space = [  Real(99.99, 99.999, name='x1'), # 硅纯度  Real(0.01, 0.1, name='x2'), # 掺杂元素A比例  Real(0.005, 0.05, name='x3'), # 掺杂元素B比例  Real(100, 500, name='x4'), # 薄膜厚度  Real(500, 1000, name='x5') # 退火温度  ]  # 贝叶斯优化目标函数  def objective\_function(params, model, X\_train, y\_train, cost\_coef=0.1):  """  目标函数：最大化(光电转化率 - 成本惩罚)  加入稳定性约束  """  x1, x2, x3, x4, x5 = params    # 稳定性约束：薄膜厚度≥200nm，退火温度≥600℃  if x4 < 200 or x5 < 600:  return -np.inf # 不稳定配方惩罚    # 成本计算与惩罚  cost = 100 + 50\*(x1 - 99.99)/0.009 + 1000\*x2 + 2000\*x3 + 0.1\*x4 + 0.05\*x5  if cost > 500: # 成本上限约束  return -np.inf    # 用随机森林预测转化率  sample = np.array([[x1, x2, x3, x4, x5]])  pred\_eff = model.predict(sample)[0]    # 加入小噪声模拟实验误差  pred\_eff += np.random.normal(0, 0.1)    # 目标值：转化率 - 成本惩罚  return pred\_eff - cost \* cost\_coef  # 主优化流程  def optimize\_material():  # 生成初始实验数据  initial\_data = generate\_material\_data(n\_initial=60)  print(f"初始实验数据规模: {initial\_data.shape}")  print(f"初始最高转化率: {initial\_data['efficiency'].max():.2f}%")  print(f"初始最低成本: {initial\_data['cost'].min():.2f}元/片")    # 划分特征与目标  X\_train = initial\_data[['x1', 'x2', 'x3', 'x4', 'x5']].values  y\_train = initial\_data['efficiency'].values    # 训练随机森林代理模型  rf\_model = RandomForestRegressor(n\_estimators=100, random\_state=42)  rf\_model.fit(X\_train, y\_train)    # 交叉验证评估模型性能  cv\_scores = cross\_val\_score(rf\_model, X\_train, y\_train, cv=5, scoring='neg\_mean\_absolute\_error')  print(f"随机森林交叉验证MAE: {np.mean(-cv\_scores):.3f}%")    # 定义带参数的目标函数  @use\_named\_args(param\_space)  def objective(\*\*params):  return -objective\_function(list(params.values()), rf\_model, X\_train, y\_train)    # 执行贝叶斯优化  print("\n开始贝叶斯优化...")  result = gp\_minimize(  objective,  param\_space,  n\_calls=40, # 额外实验次数  random\_state=42,  verbose=True,  n\_random\_starts=10 # 初始随机采样次数  )    # 提取最优配方  best\_params = result.x  best\_efficiency = -result.fun + 0.1 \* (100 + 50\*(best\_params[0] - 99.99)/0.009 +  1000\*best\_params[1] + 2000\*best\_params[2] +  0.1\*best\_params[3] + 0.05\*best\_params[4])    # 计算最优配方的成本和稳定性  best\_cost = 100 + 50\*(best\_params[0] - 99.99)/0.009 + 1000\*best\_params[1] + 2000\*best\_params[2] + 0.1\*best\_params[3] + 0.05\*best\_params[4]  best\_stability = 0.95 + 0.02\*(best\_params[3] > 300) + 0.02\*(best\_params[4] > 700) # 估计稳定性    print("\n优化结果:")  print(f"最优光电转化率: {best\_efficiency:.2f}%")  print(f"最优配方成本: {best\_cost:.2f}元/片")  print(f"估计稳定性: {best\_stability:.3f}")  print("最优配方参数:")  print(f"硅纯度: {best\_params[0]:.4f}%")  print(f"掺杂元素A比例: {best\_params[1]:.4f}%")  print(f"掺杂元素B比例: {best\_params[2]:.4f}%")  print(f"薄膜厚度: {best\_params[3]:.1f}nm")  print(f"退火温度: {best\_params[4]:.1f}℃")    # 可视化优化过程  plot\_results(initial\_data, result, best\_params, rf\_model)    return best\_params, best\_efficiency, best\_cost  # 结果可视化  def plot\_results(initial\_data, optimization\_result, best\_params, model):  """可视化优化过程和结果"""  plt.figure(figsize=(15, 12))    # 1. 优化收敛曲线  plt.subplot(2, 2, 1)  plot\_convergence(optimization\_result)  plt.title('贝叶斯优化收敛曲线')    # 2. 初始数据转化率分布  plt.subplot(2, 2, 2)  sns.histplot(initial\_data['efficiency'], kde=True)  plt.axvline(x=initial\_data['efficiency'].max(), color='r', linestyle='--', label=f'初始最大值: {initial\_data["efficiency"].max():.2f}%')  plt.axvline(x=-optimization\_result.fun + 0.1\*500, color='g', linestyle='--', label=f'优化后值: {(-optimization\_result.fun + 0.1\*500):.2f}%')  plt.title('初始实验转化率分布')  plt.xlabel('光电转化率（%）')  plt.legend()    # 3. 参数相关性热图  plt.subplot(2, 2, 3)  corr = initial\_data[['x1', 'x2', 'x3', 'x4', 'x5', 'efficiency', 'cost']].corr()  sns.heatmap(corr, annot=True, cmap='coolwarm', fmt='.2f')  plt.title('参数相关性热图')    # 4. 最优配方与初始最优对比  plt.subplot(2, 2, 4)  initial\_best\_idx = initial\_data['efficiency'].idxmax()  initial\_best = initial\_data.iloc[initial\_best\_idx]    # 转换参数到相同尺度以便对比  params\_names = ['硅纯度', '掺杂A比例', '掺杂B比例', '薄膜厚度', '退火温度']  norm\_initial = [(initial\_best[col] - param\_space[i].low)/(param\_space[i].high - param\_space[i].low)  for i, col in enumerate(['x1', 'x2', 'x3', 'x4', 'x5'])]  norm\_best = [(best\_params[i] - param\_space[i].low)/(param\_space[i].high - param\_space[i].low)  for i in range(5)]    x = np.arange(len(params\_names))  width = 0.35  plt.bar(x - width/2, norm\_initial, width, label='初始最优')  plt.bar(x + width/2, norm\_best, width, label='优化后最优')  plt.xticks(x, params\_names, rotation=30)  plt.title('最优配方参数对比（归一化）')  plt.ylabel('归一化值（0-1）')  plt.legend()    plt.tight\_layout()  plt.show()  # 执行优化  if \_\_name\_\_ == "\_\_main\_\_":  optimize\_material()  ### 10. 蚁群算法（ACO）+ 粒子群优化（PSO）求解智能物流路径规划  ```python  import numpy as np  import matplotlib.pyplot as plt  import random  import seaborn as sns  from matplotlib.patches import Circle  # 设置随机种子，保证结果可复现  np.random.seed(42)  random.seed(42)  # 问题数据初始化  def initialize\_logistics\_data(n\_customers=20, n\_vehicles=3, vehicle\_capacity=500):  """初始化物流配送问题数据"""  # 配送中心坐标  depot = np.array([50, 50])    # 客户坐标（随机分布在10-90范围内）  customers = np.random.rand(n\_customers, 2) \* 80 + 10    # 客户需求（10-50单位）  demands = np.random.randint(10, 51, size=n\_customers)    # 时间窗口（[最早时间, 最晚时间]）  time\_windows = []  for i in range(n\_customers):  start = random.randint(60, 240) # 1-4小时  end = start + random.randint(60, 180) # 持续1-3小时  time\_windows.append([start, end])    # 服务时间（5-15分钟）  service\_times = np.random.randint(5, 16, size=n\_customers)    return {  'depot': depot,  'customers': customers,  'demands': demands,  'time\_windows': np.array(time\_windows),  'service\_times': service\_times,  'n\_customers': n\_customers,  'n\_vehicles': n\_vehicles,  'vehicle\_capacity': vehicle\_capacity,  'vehicle\_speed': 0.5 # 单位距离所需时间（分钟）  }  # 距离计算  def calculate\_distance(point1, point2):  """计算两点间欧氏距离"""  return np.sqrt(np.sum((point1 - point2)\*\* 2))  # 路径成本计算  def route\_cost(route, data):  """计算单条路径的总成本（距离+时间惩罚）"""  if not route:  return 0    # 路径点坐标（包括起点和终点）  points = [data['depot']] + [data['customers'][i] for i in route] + [data['depot']]  total\_distance = 0  total\_time = 0  time\_penalty = 0  current\_time = 0    # 计算距离和时间惩罚  for i in range(len(points) - 1):  # 距离成本  dist = calculate\_distance(points[i], points[i+1])  total\_distance += dist    # 时间计算  travel\_time = dist / data['vehicle\_speed']  current\_time += travel\_time    # 如果是客户点，检查时间窗口  if i > 0 and i < len(points) - 1:  customer\_idx = route[i-1]  # 早到惩罚  if current\_time < data['time\_windows'][customer\_idx][0]:  wait\_time = data['time\_windows'][customer\_idx][0] - current\_time  time\_penalty += wait\_time \* 0.5 # 早到惩罚系数  current\_time = data['time\_windows'][customer\_idx][0]  # 迟到惩罚（更严重）  elif current\_time > data['time\_windows'][customer\_idx][1]:  delay\_time = current\_time - data['time\_windows'][customer\_idx][1]  time\_penalty += delay\_time \* 2 # 迟到惩罚系数    # 加上服务时间  current\_time += data['service\_times'][customer\_idx]    # 总时间惩罚加入总成本  total\_cost = total\_distance + time\_penalty \* 0.1  return total\_cost, total\_distance, time\_penalty  # 蚁群算法（ACO）组件  class AntColonyOptimizer:  def \_\_init\_\_(self, data, n\_ants=30, alpha=1.0, beta=2.0, rho=0.1, Q=100):  self.data = data  self.n\_ants = n\_ants  self.alpha = alpha # 信息素重要性  self.beta = beta # 启发式信息重要性  self.rho = rho # 信息素蒸发率  self.Q = Q # 信息素增量常数    # 初始化信息素矩阵  n = data['n\_customers']  self.pherom</doubaocanvas> |

### 10. 禁忌搜索（TS）+ 0-1 整数规划（IP）组合模型案例题目

**题目：农村 5G 基站选址与覆盖优化问题**

* **问题背景**：某通信公司计划在某县的 30 个行政村部署 5G 基站，每个基站覆盖半径 1.5km，建设成本 50 万元 / 个。部分村庄人口少（≤500 人）但位于交通要道，需优先覆盖；部分村庄因地形（如山区）需多基站重叠覆盖。
* **问题描述**：需选择基站建设地点（从 50 个候选点中选取），目标包括：① 覆盖所有行政村（覆盖率 100%）；② 最小化建设总成本；③ 最大化人口覆盖率（≥95%）。变量为 0-1 型（1 表示建设，0 表示不建设），需满足山区村庄至少 2 个基站覆盖的约束。
* **数据情况**：提供各村的位置坐标、人口数量、地形类型（平原 / 山区），候选基站的位置坐标、建设成本、预计覆盖范围（受地形影响的修正系数），以及村庄之间的道路连接情况（影响信号传播）。

### 10. 禁忌搜索（TS）+ 0-1 整数规划（IP）求解农村 5G 基站选址与覆盖优化代码

|  |
| --- |
| import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  import seaborn as sns  import random  import pulp  from matplotlib.patches import Circle, Polygon  from matplotlib.colors import LinearSegmentedColormap  import copy  # 设置随机种子，保证结果可复现  np.random.seed(42)  random.seed(42)  # 问题数据初始化  def initialize\_rural\_5g\_data(n\_villages=30, n\_candidates=15, coverage\_radius=3.0):  """  初始化农村5G基站选址问题数据  n\_villages: 村庄数量  n\_candidates: 候选基站位置数量  coverage\_radius: 基站覆盖半径（公里）  """  # 生成村庄坐标（模拟农村区域分布）  # 假设区域范围为10x10平方公里  villages = np.random.rand(n\_villages, 2) \* 10    # 村庄人口（影响优先级）  population = np.random.randint(100, 1501, size=n\_villages) # 100-1500人    # 村庄经济水平（1-5，影响权重）  economic\_level = np.random.randint(1, 6, size=n\_villages)    # 地形复杂度（影响建设成本和信号衰减）  # 0-平坦，1-丘陵，2-山区  terrain = np.random.choice([0, 1, 2], size=n\_villages, p=[0.5, 0.3, 0.2])    # 候选基站位置（尽量选择地势较高处）  candidates = np.random.rand(n\_candidates, 2) \* 10    # 基站建设成本（受地形影响）  base\_cost = 150 # 基础建设成本（万元）  build\_cost = []  for i in range(n\_candidates):  # 计算候选点周围地形影响  nearby\_terrain = []  for j in range(n\_villages):  dist = np.sqrt(np.sum((candidates[i] - villages[j])\*\*2))  if dist < coverage\_radius:  nearby\_terrain.append(terrain[j])    if not nearby\_terrain:  terrain\_factor = 1.0  else:  # 地形越复杂，建设成本越高  terrain\_factor = 1.0 + 0.3 \* np.mean(nearby\_terrain)    # 最终建设成本  cost = base\_cost \* terrain\_factor + np.random.normal(0, 10)  build\_cost.append(max(100, cost)) # 最低100万元    # 计算每个候选基站对村庄的覆盖情况（考虑地形衰减）  coverage = np.zeros((n\_candidates, n\_villages)) # 1表示可覆盖，0表示不可  signal\_strength = np.zeros((n\_candidates, n\_villages)) # 信号强度（0-1）    for i in range(n\_candidates):  for j in range(n\_villages):  # 计算距离  dist = np.sqrt(np.sum((candidates[i] - villages[j])\*\*2))    # 距离衰减  if dist <= coverage\_radius:  # 距离越近信号越强  dist\_factor = 1 - (dist / coverage\_radius)    # 地形衰减  terrain\_loss = {0: 0.1, 1: 0.3, 2: 0.5}[terrain[j]]    # 最终信号强度  strength = dist\_factor \* (1 - terrain\_loss)  signal\_strength[i, j] = strength    # 信号强度大于0.3视为有效覆盖  if strength > 0.3:  coverage[i, j] = 1    # 基站间干扰系数（距离越近干扰越大）  interference = np.zeros((n\_candidates, n\_candidates))  for i in range(n\_candidates):  for j in range(n\_candidates):  if i != j:  dist = np.sqrt(np.sum((candidates[i] - candidates[j])\*\*2))  # 距离越近，干扰系数越大（0-1）  interference[i, j] = max(0, 1 - dist / (2 \* coverage\_radius))    return {  'villages': villages,  'population': population,  'economic\_level': economic\_level,  'terrain': terrain,  'candidates': candidates,  'build\_cost': np.array(build\_cost),  'coverage': coverage,  'signal\_strength': signal\_strength,  'interference': interference,  'coverage\_radius': coverage\_radius,  'n\_villages': n\_villages,  'n\_candidates': n\_candidates,  'annual\_operation\_cost': 15 # 年运维成本（万元/基站）  }  # 0-1整数规划模型（固定基站集合下的优化）  def ip\_coverage\_optimization(data, selected\_candidates):  """  在选定的候选基站集合中，使用0-1整数规划优化最终选址  目标：最小化总成本，最大化覆盖质量  """  n\_selected = len(selected\_candidates)  n\_villages = data['n\_villages']    if n\_selected == 0:  return [], float('inf')    # 创建问题实例（最小化成本）  prob = pulp.LpProblem("5G\_Base\_Station\_Optimization", pulp.LpMinimize)    # 决策变量：是否建设基站（1-建设，0-不建设）  x = pulp.LpVariable.dicts(  "Build", range(n\_selected), lowBound=0, upBound=1, cat=pulp.LpInteger  )    # 辅助变量：村庄是否被覆盖  y = pulp.LpVariable.dicts(  "Covered", range(n\_villages), lowBound=0, upBound=1, cat=pulp.LpInteger  )    # 1. 目标函数：最小化建设成本 + 未覆盖惩罚  total\_cost = 0    # 建设成本  for i in range(n\_selected):  idx = selected\_candidates[i]  total\_cost += x[i] \* data['build\_cost'][idx]    # 未覆盖惩罚（基于村庄重要性）  # 重要性 = 人口 \* 经济水平  importance = data['population'] \* data['economic\_level']  max\_importance = np.max(importance)    for j in range(n\_villages):  # 惩罚系数：相当于覆盖100人的成本  penalty = max\_importance \* 0.1  total\_cost += (1 - y[j]) \* penalty    prob += total\_cost, "Total\_Cost"    # 2. 约束条件  # 2.1 覆盖约束：至少一个基站覆盖村庄  for j in range(n\_villages):  coverage\_expr = 0  for i in range(n\_selected):  candidate\_idx = selected\_candidates[i]  coverage\_expr += x[i] \* data['coverage'][candidate\_idx, j]    # 如果有基站覆盖，则y[j]必须为1  prob += coverage\_expr >= y[j], f"Coverage\_Constraint\_{j}"    # 2.2 干扰约束：避免过近的基站同时建设  for i in range(n\_selected):  for k in range(i+1, n\_selected):  i\_idx = selected\_candidates[i]  k\_idx = selected\_candidates[k]    # 干扰系数大于0.5的基站不能同时建设  if data['interference'][i\_idx, k\_idx] > 0.5:  prob += x[i] + x[k] <= 1, f"Interference\_Constraint\_{i}\_{k}"    # 2.3 预算约束（假设总预算）  total\_budget = 1200 # 总预算（万元）  prob += pulp.lpSum([x[i] \* data['build\_cost'][selected\_candidates[i]] for i in range(n\_selected)]) <= total\_budget, "Budget\_Constraint"    # 求解  prob.solve(pulp.PULP\_CBC\_CMD(msg=0)) # 静音模式    # 提取结果  selected = [selected\_candidates[i] for i in range(n\_selected) if x[i].varValue > 0.9]  total\_cost = pulp.value(prob.objective)    return selected, total\_cost  # 禁忌搜索算法  class TabuSearch:  def \_\_init\_\_(self, data, tabu\_size=10, max\_iter=50, neighbor\_size=5):  """  禁忌搜索算法初始化  tabu\_size: 禁忌表大小  max\_iter: 最大迭代次数  neighbor\_size: 每次生成的邻域解数量  """  self.data = data  self.tabu\_size = tabu\_size  self.max\_iter = max\_iter  self.neighbor\_size = neighbor\_size    self.n\_candidates = data['n\_candidates']  self.tabu\_list = [] # 禁忌表，存储被禁忌的操作  self.best\_solution = []  self.best\_cost = float('inf')  self.history = [] # 记录优化历史    def initialize\_solution(self):  """生成初始解：随机选择部分候选基站"""  # 初始选择30%-50%的候选点  n\_selected = int(self.n\_candidates \* (0.3 + 0.2 \* random.random()))  return random.sample(range(self.n\_candidates), n\_selected)    def evaluate\_solution(self, solution):  """评估解的质量：调用整数规划模型"""  if not solution:  return float('inf')    # 调用IP模型进行优化  optimized\_sol, cost = ip\_coverage\_optimization(self.data, solution)  return optimized\_sol, cost    def generate\_neighbors(self, current\_solution):  """生成邻域解"""  neighbors = []    # 当前选中的基站集合  current\_set = set(current\_solution)  # 未选中的基站集合  unselected = set(range(self.n\_candidates)) - current\_set    # 生成多种邻域解  for \_ in range(self.neighbor\_size):  # 随机选择操作类型：添加、删除或替换  operation = random.choice(['add', 'remove', 'swap'])    if operation == 'add' and unselected:  # 添加一个基站  to\_add = random.choice(list(unselected))  new\_sol = current\_solution + [to\_add]    elif operation == 'remove' and len(current\_solution) > 1:  # 删除一个基站  to\_remove = random.choice(current\_solution)  new\_sol = [s for s in current\_solution if s != to\_remove]    elif operation == 'swap' and unselected and current\_solution:  # 替换一个基站  to\_remove = random.choice(current\_solution)  to\_add = random.choice(list(unselected))  new\_sol = [s for s in current\_solution if s != to\_remove] + [to\_add]    else:  # 无法执行所选操作，默认选择swap  if unselected and current\_solution:  to\_remove = random.choice(current\_solution)  to\_add = random.choice(list(unselected))  new\_sol = [s for s in current\_solution if s != to\_remove] + [to\_add]  elif unselected:  to\_add = random.choice(list(unselected))  new\_sol = current\_solution + [to\_add]  else:  to\_remove = random.choice(current\_solution)  new\_sol = [s for s in current\_solution if s != to\_remove]    # 去重并排序（便于比较）  new\_sol = sorted(list(set(new\_sol)))  neighbors.append(new\_sol)    return neighbors    def is\_tabu(self, solution, current\_solution):  """判断解是否为禁忌"""  # 计算与当前解的差异（操作）  current\_set = set(current\_solution)  new\_set = set(solution)    added = new\_set - current\_set  removed = current\_set - new\_set    # 操作表示为(添加的基站, 删除的基站)  operation = (tuple(sorted(added)), tuple(sorted(removed)))    return operation in self.tabu\_list    def update\_tabu\_list(self, solution, current\_solution):  """更新禁忌表"""  current\_set = set(current\_solution)  new\_set = set(solution)    added = new\_set - current\_set  removed = current\_set - new\_set    operation = (tuple(sorted(added)), tuple(sorted(removed)))    # 添加新操作到禁忌表  self.tabu\_list.append(operation)    # 如果超出禁忌表大小，移除最早的  if len(self.tabu\_list) > self.tabu\_size:  self.tabu\_list.pop(0)    def optimize(self):  """执行禁忌搜索优化"""  # 初始解  current\_solution = self.initialize\_solution()  current\_solution, current\_cost = self.evaluate\_solution(current\_solution)    # 更新最优解  self.best\_solution = current\_solution  self.best\_cost = current\_cost  self.history.append(self.best\_cost)    print(f"初始解: 选择{len(current\_solution)}个基站, 成本{current\_cost:.2f}万元")    # 迭代优化  for iter in range(self.max\_iter):  # 生成邻域解  neighbors = self.generate\_neighbors(current\_solution)    # 评估所有邻域解  neighbor\_results = []  for neighbor in neighbors:  # 避免重复解  if neighbor == current\_solution:  continue    # 评估邻域解  opt\_neighbor, cost = self.evaluate\_solution(neighbor)  neighbor\_results.append((neighbor, opt\_neighbor, cost))    # 按成本排序  neighbor\_results.sort(key=lambda x: x[2])    # 选择下一个解  next\_solution = None  next\_opt\_solution = None  next\_cost = float('inf')    for nr in neighbor\_results:  neighbor, opt\_neighbor, cost = nr    # 检查是否为禁忌解  if not self.is\_tabu(neighbor, current\_solution):  next\_solution = neighbor  next\_opt\_solution = opt\_neighbor  next\_cost = cost  break  else:  # 特赦准则：如果禁忌解明显优于当前最优解，则接受  if cost < self.best\_cost \* 0.9:  next\_solution = neighbor  next\_opt\_solution = opt\_neighbor  next\_cost = cost  print(f"触发特赦准则: 成本{cost:.2f}优于当前最优{self.best\_cost:.2f}")  break    # 如果没有找到合适的解，随机选择一个  if next\_solution is None and neighbor\_results:  next\_solution, next\_opt\_solution, next\_cost = neighbor\_results[0]    # 更新当前解  current\_solution = next\_solution  current\_cost = next\_cost    # 更新最优解  if current\_cost < self.best\_cost:  self.best\_solution = next\_opt\_solution  self.best\_cost = current\_cost  print(f"迭代{iter+1}: 找到更优解, 选择{len(self.best\_solution)}个基站, 成本{self.best\_cost:.2f}万元")    # 更新禁忌表  self.update\_tabu\_list(current\_solution, [s for s in current\_solution if s not in next\_solution or next\_solution])    # 记录历史  self.history.append(self.best\_cost)    print(f"优化完成: 最优解选择{len(self.best\_solution)}个基站, 总成本{self.best\_cost:.2f}万元")  return self.best\_solution, self.best\_cost  # 结果分析与可视化  def analyze\_and\_visualize(data, best\_solution):  """分析并可视化基站选址结果"""  # 1. 计算覆盖情况  covered\_villages = set()  coverage\_detail = []    for station in best\_solution:  for village in range(data['n\_villages']):  if data['coverage'][station, village] == 1:  covered\_villages.add(village)  coverage\_detail.append({  'village': village,  'station': station,  'signal': data['signal\_strength'][station, village]  })    coverage\_rate = len(covered\_villages) / data['n\_villages'] \* 100  print(f"村庄覆盖率: {coverage\_rate:.2f}%")    # 2. 计算建设总成本  total\_build\_cost = sum(data['build\_cost'][s] for s in best\_solution)  print(f"总建设成本: {total\_build\_cost:.2f}万元")  print(f"年均</doubaocanvas> |

### 11. 人工蜂群算法（ABC）+ 差分进化（DE）组合模型案例题目

**题目：工业机器人焊接路径优化问题**

* **问题背景**：某汽车零部件工厂的焊接机器人需对变速箱壳体的 20 个焊点进行焊接，当前路径存在重复移动（空行程占比 30%），导致单件焊接时间达 8 分钟，制约产能。机器人运动时需避障（如夹具、凸起结构），最小转弯半径 0.5m。
* **问题描述**：需优化焊接点的访问顺序和运动路径，目标包括：① 最小化总运动时间（≤5 分钟）；② 最小化路径长度；③ 确保相邻焊点的运动轨迹平滑（转弯角度≤90°）。需处理机器人运动学的非线性约束。
* **数据情况**：提供各焊点的三维坐标（x,y,z）、机器人当前运动路径记录、各轴运动速度参数（m/s）、障碍物的位置和尺寸、不同路径下的实际焊接时间。

### 11. 人工蜂群算法（ABC）+ 差分进化（DE）求解工业机器人焊接路径优化代码

|  |
| --- |
| import numpy as np  import matplotlib.pyplot as plt  import random  import copy  from mpl\_toolkits.mplot3d import Axes3D  from matplotlib.patches import Circle, PathPatch  from matplotlib.path import Path  import time  # 设置随机种子，保证结果可复现  np.random.seed(42)  random.seed(42)  # 问题数据初始化  def initialize\_welding\_data(n\_points=25, workspace\_size=(1000, 800, 500)):  """  初始化工业机器人焊接路径优化问题数据  n\_points: 焊接点数量  workspace\_size: 工作空间尺寸 (x, y, z)，单位：毫米  """  # 生成焊接点坐标（3D空间）  x\_range, y\_range, z\_range = workspace\_size  points = np.zeros((n\_points, 3))    # 模拟工件表面的焊接点分布（假设为不规则曲面）  # 基础曲面：轻微弯曲的平面  base\_curve = np.linspace(0, x\_range, n\_points)  z\_base = 100 + 50 \* np.sin(base\_curve / x\_range \* 2 \* np.pi) # 基础高度    for i in range(n\_points):  points[i, 0] = random.uniform(0, x\_range) # x坐标  points[i, 1] = random.uniform(0, y\_range) # y坐标  points[i, 2] = z\_base[i] + random.uniform(-30, 30) # z坐标（在曲面上波动）    # 焊接点属性  # 1. 焊接难度系数（1-5，影响焊接时间）  difficulty = np.random.randint(1, 6, size=n\_points)    # 2. 焊接精度要求（1-5，影响路径平滑度需求）  precision = np.random.randint(1, 6, size=n\_points)    # 3. 热变形系数（相邻点焊接顺序影响变形量）  heat\_deformation = np.zeros((n\_points, n\_points))  for i in range(n\_points):  for j in range(n\_points):  if i != j:  # 距离越近，热变形影响越大  dist = np.sqrt(np.sum((points[i] - points[j])\*\*2))  heat\_deformation[i, j] = max(0, 1 - dist / 500) # 500mm为影响阈值    # 4. 机器人移动参数  robot\_params = {  'max\_speed': 500, # 最大移动速度（mm/s）  'acceleration': 200, # 加速度（mm/s²）  'tool\_radius': 15, # 焊枪半径（mm）  'min\_distance': 10 # 最小安全距离（mm）  }    # 5. 焊接工艺参数  welding\_params = {  'base\_time': 2.0, # 基础焊接时间（秒）  'speed\_factor': 0.005, # 移动时间系数  'precision\_factor': 0.3 # 精度对时间的影响系数  }    return {  'points': points,  'difficulty': difficulty,  'precision': precision,  'heat\_deformation': heat\_deformation,  'robot\_params': robot\_params,  'welding\_params': welding\_params,  'n\_points': n\_points,  'workspace\_size': workspace\_size  }  # 路径成本计算  def calculate\_path\_cost(path, data):  """  计算焊接路径的总成本  path: 焊接点索引的有序列表  成本包括：移动时间 + 焊接时间 + 热变形惩罚  """  if len(path) <= 1:  return 0.0    points = data['points']  n\_points = data['n\_points']  robot = data['robot\_params']  welding = data['welding\_params']    total\_cost = 0.0  total\_move\_time = 0.0  total\_weld\_time = 0.0  deformation\_penalty = 0.0    # 计算移动时间和焊接时间  for i in range(len(path)):  # 焊接时间（与难度和精度相关）  point\_idx = path[i]  weld\_time = welding['base\_time'] \* data['difficulty'][point\_idx] \* (1 + 0.1 \* data['precision'][point\_idx])  total\_weld\_time += weld\_time    # 移动时间（到下一个点）  if i < len(path) - 1:  next\_idx = path[i+1]  dist = np.sqrt(np.sum((points[point\_idx] - points[next\_idx])\*\*2))    # 计算移动时间（考虑加速到最大速度）  # 加速阶段距离：v²/(2a)  max\_speed = robot['max\_speed']  acceleration = robot['acceleration']  加速距离 = (max\_speed\*\*2) / (2 \* acceleration)    if dist <= 2 \* 加速距离:  # 未达到最大速度就需要减速  time = 2 \* np.sqrt(dist / acceleration)  else:  # 达到最大速度后匀速运动  time = (2 \* 加速距离) / max\_speed + (dist - 2 \* 加速距离) / max\_speed    total\_move\_time += time    # 热变形惩罚（基于焊接顺序）  deformation\_penalty += data['heat\_deformation'][point\_idx][next\_idx] \* 2.0 # 惩罚系数    # 总路径成本：时间（秒）+ 惩罚  total\_cost = total\_move\_time + total\_weld\_time + deformation\_penalty    return total\_cost, {  'move\_time': total\_move\_time,  'weld\_time': total\_weld\_time,  'deformation': deformation\_penalty  }  # 差分进化（DE）算法组件  class DifferentialEvolution:  def \_\_init\_\_(self, data, pop\_size=30, f=0.5, cr=0.7, max\_iter=50):  """  差分进化算法初始化  pop\_size: 种群大小  f: 缩放因子  cr: 交叉概率  max\_iter: 最大迭代次数  """  self.data = data  self.pop\_size = pop\_size  self.f = f # 缩放因子  self.cr = cr # 交叉概率  self.max\_iter = max\_iter  self.n\_points = data['n\_points']    # 初始化种群  self.population = []  self.fitness = []    # 生成初始路径（随机排列）  for \_ in range(pop\_size):  path = list(np.random.permutation(self.n\_points))  self.population.append(path)  cost, \_ = calculate\_path\_cost(path, data)  self.fitness.append(cost)    # 记录最优解  self.best\_idx = np.argmin(self.fitness)  self.best\_path = self.population[self.best\_idx]  self.best\_cost = self.fitness[self.best\_idx]  self.history = [self.best\_cost]    def mutate(self, idx):  """变异操作：生成变异向量"""  # 随机选择3个不同的个体  idxs = list(range(self.pop\_size))  idxs.remove(idx)  a, b, c = random.sample(idxs, 3)    # 生成变异向量  mutant = []  for i in range(self.n\_points):  # 基于位置的差分变异  val = self.population[c][i] + self.f \* (self.population[a][i] - self.population[b][i])  # 确保在有效范围内  val = int(round(np.clip(val, 0, self.n\_points - 1)))  mutant.append(val)    # 修复重复元素（保持排列特性）  return self.\_repair\_duplicates(mutant)    def crossover(self, target, mutant):  """交叉操作：生成试验向量"""  trial = []  for i in range(self.n\_points):  if random.random() < self.cr or i == random.randint(0, self.n\_points - 1):  trial.append(mutant[i])  else:  trial.append(target[i])    # 修复重复元素  return self.\_repair\_duplicates(trial)    def \_repair\_duplicates(self, vector):  """修复向量中的重复元素，确保是有效排列"""  seen = set()  duplicates = []  new\_vector = []    for val in vector:  if val not in seen:  seen.add(val)  new\_vector.append(val)  else:  duplicates.append(val)    # 补充缺失的元素  missing = [x for x in range(self.n\_points) if x not in seen]  for i in range(len(duplicates)):  new\_vector.append(missing[i])    return new\_vector    def select(self, trial, idx):  """选择操作：保留较优个体"""  trial\_cost, \_ = calculate\_path\_cost(trial, self.data)  if trial\_cost < self.fitness[idx]:  self.population[idx] = trial  self.fitness[idx] = trial\_cost    # 更新全局最优  if trial\_cost < self.best\_cost:  self.best\_path = trial  self.best\_cost = trial\_cost    def optimize(self):  """执行差分进化优化"""  print(f"DE初始最优成本: {self.best\_cost:.2f}秒")    for iter in range(self.max\_iter):  for i in range(self.pop\_size):  # 变异  mutant = self.mutate(i)  # 交叉  trial = self.crossover(self.population[i], mutant)  # 选择  self.select(trial, i)    # 记录历史  self.history.append(self.best\_cost)    # 定期输出  if (iter + 1) % 10 == 0:  print(f"DE迭代{iter+1}/{self.max\_iter}, 最优成本: {self.best\_cost:.2f}秒")    return self.best\_path, self.best\_cost  # 人工蜂群算法（ABC）组件  class ArtificialBeeColony:  def \_\_init\_\_(self, data, de\_solution=None, colony\_size=30, limit=20, max\_iter=50):  """  人工蜂群算法初始化  colony\_size: 蜂群大小  limit: 放弃食物源的限制次数  max\_iter: 最大迭代次数  de\_solution: 差分进化得到的初始解（用于初始化）  """  self.data = data  self.colony\_size = colony\_size  self.limit = limit # 侦查蜂阈值  self.max\_iter = max\_iter  self.n\_points = data['n\_points']    # 雇佣蜂和观察蜂各占一半  self.employed\_bees = colony\_size // 2  self.onlooker\_bees = colony\_size - self.employed\_bees    # 初始化食物源（焊接路径）  self.food\_sources = []  self.fitness = []  self.trials = [] # 记录各食物源未改进的次数    # 如果有DE的初始解，优先使用  if de\_solution is not None:  self.food\_sources.append(de\_solution)  cost, \_ = calculate\_path\_cost(de\_solution, data)  self.fitness.append(cost)  self.trials.append(0)  # 补充其他随机解  for \_ in range(colony\_size - 1):  path = list(np.random.permutation(self.n\_points))  self.food\_sources.append(path)  cost, \_ = calculate\_path\_cost(path, data)  self.fitness.append(cost)  self.trials.append(0)  else:  # 全部随机初始化  for \_ in range(colony\_size):  path = list(np.random.permutation(self.n\_points))  self.food\_sources.append(path)  cost, \_ = calculate\_path\_cost(path, data)  self.fitness.append(cost)  self.trials.append(0)    # 记录最优解  self.best\_idx = np.argmin(self.fitness)  self.best\_path = self.food\_sources[self.best\_idx]  self.best\_cost = self.fitness[self.best\_idx]  self.history = [self.best\_cost]    def employed\_bee\_phase(self):  """雇佣蜂阶段：搜索邻域解"""  for i in range(self.employed\_bees):  current\_path = self.food\_sources[i]  current\_cost = self.fitness[i]    # 生成邻域解（随机交换两个点）  neighbor\_path = self.\_generate\_neighbor(current\_path)    # 评估新解  neighbor\_cost, \_ = calculate\_path\_cost(neighbor\_path, self.data)    # 贪婪选择  if neighbor\_cost < current\_cost:  self.food\_sources[i] = neighbor\_path  self.fitness[i] = neighbor\_cost  self.trials[i] = 0 # 重置计数    # 更新全局最优  if neighbor\_cost < self.best\_cost:  self.best\_path = neighbor\_path  self.best\_cost = neighbor\_cost  else:  self.trials[i] += 1 # 增加计数    def onlooker\_bee\_phase(self):  """观察蜂阶段：根据适应度选择食物源"""  # 计算选择概率（基于成本的倒数）  total\_fitness = sum(1.0 / cost for cost in self.fitness[:self.employed\_bees])  probabilities = [(1.0 / cost) / total\_fitness for cost in self.fitness[:self.employed\_bees]]    for \_ in range(self.onlooker\_bees):  # 轮盘赌选择食物源  idx = np.random.choice(self.employed\_bees, p=probabilities)  current\_path = self.food\_sources[idx]  current\_cost = self.fitness[idx]    # 生成邻域解（随机反转一段路径）  neighbor\_path = self.\_generate\_neighbor(current\_path, method='reverse')    # 评估新解  neighbor\_cost, \_ = calculate\_path\_cost(neighbor\_path, self.data)    # 贪婪选择  if neighbor\_cost < current\_cost:  self.food\_sources[idx] = neighbor\_path  self.fitness[idx] = neighbor\_cost  self.trials[idx] = 0 # 重置计数    # 更新全局最优  if neighbor\_cost < self.best\_cost:  self.best\_path = neighbor\_path  self.best\_cost = neighbor\_cost  else:  self.trials[idx] += 1 # 增加计数    def scout\_bee\_phase(self):  """侦查蜂阶段：替换长时间未改进的食物源"""  for i in range(self.employed\_bees):  if self.trials[i] > self.limit:  # 随机生成新解  new\_path = list(np.random.permutation(self.n\_points))  new\_cost, \_ = calculate\_path\_cost(new\_path, self.data)    # 替换旧解  self.food\_sources[i] = new\_path  self.fitness[i] = new\_cost  self.trials[i] = 0    # 可能的局部优化：应用2-opt操作  improved = True  while improved:  improved = False  for a in range(len(new\_path) - 1):  for b in range(a + 1, len(new\_path)):  if b - a == 1:  continue # 相邻点不交换  # 反转a到b之间的路径  reversed\_path = new\_path[:a] + new\_path[a:b+1][::-1] + new\_path[b+1:]  reversed\_cost, \_ = calculate\_path\_cost(reversed\_path, self.data)  if reversed\_cost < new\_cost:  new\_path = reversed\_path  new\_cost = reversed\_cost  improved = True  break  if improved:  break    # 更新解  self.food\_sources[i] = new\_path  self.fitness[i] = new\_cost    # 更新全局最优  if new\_cost < self.best\_cost:  self.best\_path = new\_path  self.best\_cost = new\_cost    def \_generate\_neighbor(self, path, method='swap'):  """生成邻域解"""  neighbor = copy.copy(path)    if method == 'swap':  # 随机交换两个点  i, j = random.sample(range(len(path)), 2)  neighbor[i], neighbor[j] = neighbor[j], neighbor[i]  elif method == 'reverse':  # 随机反转一段路径  i, j = sorted(random.sample(range(len(path)), 2))  neighbor[i:j+1] = neighbor[i:j+1][::-1]  elif method == 'insert':  # 随机插入一个点  i, j = random.sample(range(len(path</doubaocanvas> |

### 12. 模糊神经网络（FNN）+ 粒子群优化（PSO）组合模型案例题目

**题目：城市污水处理厂曝气参数优化问题**

* **问题背景**：某污水处理厂采用活性污泥法处理城市污水，曝气环节的溶解氧浓度（DO）、曝气时长直接影响 COD 去除率和能耗（曝气能耗占总能耗的 60%）。当前采用固定参数（DO=2mg/L，曝气 4 小时），COD 去除率仅 80%，且冬季因水温低效率下降更明显。
* **问题描述**：需根据进水水质（含模糊特征如 “COD 较高”“污泥浓度偏稀”）动态调整曝气参数，目标包括：① 最大化 COD 去除率（≥90%）；② 最小化曝气能耗；③ 避免 DO 浓度剧烈波动（≤±0.5mg/L）。
* **数据情况**：提供连续 6 个月的运行数据，包括：进水 COD 浓度（mg/L）、污泥浓度（g/L）、水温（℃）、当前曝气参数、COD 去除率、小时能耗（kWh）。数据中含传感器噪声（如 DO 测量误差 ±0.2mg/L）。

### 12. 模糊神经网络（FNN）+ 粒子群优化（PSO）求解城市污水处理厂曝气参数优化代码

|  |
| --- |
| import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  import seaborn as sns  import random  import copy  from sklearn.preprocessing import MinMaxScaler  from sklearn.model\_selection import train\_test\_split  from sklearn.metrics import mean\_squared\_error, r2\_score  import matplotlib.dates as mdates  from matplotlib.animation import FuncAnimation  # 设置随机种子，保证结果可复现  np.random.seed(42)  random.seed(42)  # 1. 数据生成与预处理  def generate\_sewage\_data(n\_samples=180, noise\_level=0.05):  """  生成污水处理厂运行数据（模拟6个月数据）  n\_samples: 样本数量（天）  noise\_level: 数据噪声水平  """  # 时间序列  dates = pd.date\_range(start='2023-01-01', periods=n\_samples, freq='D')    # 环境与进水参数  data = pd.DataFrame()  data['date'] = dates    # 水温（随季节变化，1-12月）  month = dates.month  data['water\_temp'] = 15 + 10 \* np.sin(month / 12 \* 2 \* np.pi) + np.random.normal(0, 1, n\_samples)  data['water\_temp'] = np.clip(data['water\_temp'], 5, 28) # 限制在5-28℃    # 进水COD浓度（100-400 mg/L）  base\_cod = 250 + 100 \* np.sin(np.linspace(0, 4\*np.pi, n\_samples))  data['influent\_cod'] = base\_cod + np.random.normal(0, 30, n\_samples)  data['influent\_cod'] = np.clip(data['influent\_cod'], 100, 400)    # 污泥浓度（2-6 g/L）  data['sludge\_conc'] = 4 + 1.5 \* np.sin(np.linspace(0, 2\*np.pi, n\_samples) + np.pi) + np.random.normal(0, 0.3, n\_samples)  data['sludge\_conc'] = np.clip(data['sludge\_conc'], 2, 6)    # 传统曝气参数（固定值）  data['do\_setpoint'] = 2.0 # 溶解氧设定值（mg/L）  data['aeration\_time'] = 4.0 # 曝气时长（小时）    # 计算COD去除率（受多种因素影响）  # 温度影响：温度升高，反应加快  temp\_factor = 0.8 + 0.2 \* (data['water\_temp'] - 5) / 23 # 归一化到0.8-1.0    # 污泥浓度影响：适中浓度最佳  sludge\_factor = 1 - 0.3 \* abs(data['sludge\_conc'] - 4) / 2 # 4g/L时最佳    # 进水COD影响：浓度过高抑制去除率  cod\_factor = 1 - 0.2 \* (data['influent\_cod'] - 250) / 150 # 250mg/L时最佳  cod\_factor = np.clip(cod\_factor, 0.6, 1.0)    # 基础去除率（传统参数下）  base\_removal = 0.8 + 0.05 \* (temp\_factor + sludge\_factor + cod\_factor - 2.4)  data['cod\_removal'] = base\_removal + np.random.normal(0, noise\_level, n\_samples)  data['cod\_removal'] = np.clip(data['cod\_removal'], 0.6, 0.9) # 限制在60%-90%    # 计算能耗（kWh）  # 曝气能耗与曝气时间、DO设定值正相关  energy\_base = 150 + data['aeration\_time'] \* 20 + data['do\_setpoint'] \* 30  # 低温时需更高能耗  temp\_energy\_factor = 1 + (15 - data['water\_temp']) / 30  data['energy\_consumption'] = energy\_base \* temp\_energy\_factor + np.random.normal(0, 10, n\_samples)  data['energy\_consumption'] = np.clip(data['energy\_consumption'], 100, 400)    # DO波动（mg/L）  data['do\_fluctuation'] = 0.3 + 0.2 \* np.random.random(n\_samples) # 传统控制下的波动    return data  # 2. 模糊逻辑系统设计  class FuzzySystem:  def \_\_init\_\_(self):  """初始化模糊逻辑系统，定义模糊集和规则"""  # 输入变量：进水COD、污泥浓度、水温  # 输出变量：DO设定值调整量、曝气时间调整量    # 2.1 定义模糊集  # 进水COD模糊集：低(L)、中(M)、高(H)  self.cod\_mf = {  'L': {'center': 150, 'width': 80}, # 梯形隶属度函数参数  'M': {'center': 250, 'width': 100},  'H': {'center': 350, 'width': 80}  }    # 污泥浓度模糊集：稀(L)、适中(M)、浓(H)  self.sludge\_mf = {  'L': {'center': 2.5, 'width': 1.0},  'M': {'center': 4.0, 'width': 1.0},  'H': {'center': 5.5, 'width': 1.0}  }    # 水温模糊集：低(L)、中(M)、高(H)  self.temp\_mf = {  'L': {'center': 10, 'width': 5},  'M': {'center': 18, 'width': 6},  'H': {'center': 25, 'width': 5}  }    # 输出：DO调整量模糊集：负大(NB)、负小(NS)、零(Z)、正小(PS)、正大(PB)  self.do\_delta\_mf = {  'NB': {'center': -0.8, 'width': 0.5},  'NS': {'center': -0.3, 'width': 0.4},  'Z': {'center': 0, 'width': 0.3},  'PS': {'center': 0.3, 'width': 0.4},  'PB': {'center': 0.8, 'width': 0.5}  }    # 输出：曝气时间调整量模糊集：负大(NB)、负小(NS)、零(Z)、正小(PS)、正大(PB)  self.time\_delta\_mf = {  'NB': {'center': -1.5, 'width': 0.8},  'NS': {'center': -0.5, 'width': 0.6},  'Z': {'center': 0, 'width': 0.5},  'PS': {'center': 0.5, 'width': 0.6},  'PB': {'center': 1.5, 'width': 0.8}  }    # 2.2 定义模糊规则（共3×3×3=27条规则）  self.rules = []    # 规则生成：根据专家知识  # 规则格式：(cod, sludge, temp) → (do\_delta, time\_delta)  cod\_levels = ['L', 'M', 'H']  sludge\_levels = ['L', 'M', 'H']  temp\_levels = ['L', 'M', 'H']    for c in cod\_levels:  for s in sludge\_levels:  for t in temp\_levels:  # 确定DO调整量  if c == 'H': # COD高需要更高DO  do = 'PB' if t == 'L' else 'PS'  elif c == 'L': # COD低可以降低DO  do = 'NB' if t == 'H' else 'NS'  else: # COD中等  if s == 'H': # 污泥浓，需更高DO  do = 'PS'  elif s == 'L': # 污泥稀，需较低DO  do = 'NS'  else: # 污泥适中  do = 'Z'    # 确定曝气时间调整量  if c == 'H': # COD高需要更长曝气时间  time = 'PB' if t == 'L' else 'PS'  elif c == 'L': # COD低可以缩短曝气时间  time = 'NB' if t == 'H' else 'NS'  else: # COD中等  if t == 'L': # 温度低需要更长时间  time = 'PS'  elif t == 'H': # 温度高可以缩短时间  time = 'NS'  else: # 温度适中  time = 'Z'    self.rules.append( ((c, s, t), (do, time)) )    def membership(self, x, center, width):  """梯形隶属度函数"""  if x < center - width:  return 0.0  elif center - width <= x < center:  return (x - (center - width)) / width  elif center <= x < center + width:  return (center + width - x) / width  else:  return 0.0    def fuzzify(self, cod, sludge, temp):  """模糊化输入变量"""  # 计算各输入的隶属度  cod\_degree = {k: self.membership(cod, v['center'], v['width'])  for k, v in self.cod\_mf.items()}  sludge\_degree = {k: self.membership(sludge, v['center'], v['width'])  for k, v in self.sludge\_mf.items()}  temp\_degree = {k: self.membership(temp, v['center'], v['width'])  for k, v in self.temp\_mf.items()}    return cod\_degree, sludge\_degree, temp\_degree    def infer(self, cod\_degree, sludge\_degree, temp\_degree):  """模糊推理"""  do\_weights = {k: 0.0 for k in self.do\_delta\_mf.keys()}  time\_weights = {k: 0.0 for k in self.time\_delta\_mf.keys()}    total\_weight = 0.0    for (cond, res), rule in enumerate(self.rules):  (c, s, t), (do, time) = rule    # 规则触发强度（取最小值）  strength = min(cod\_degree[c], sludge\_degree[s], temp\_degree[t])    if strength > 0:  # 累积权重  do\_weights[do] += strength  time\_weights[time] += strength  total\_weight += strength    return do\_weights, time\_weights, total\_weight    def defuzzify(self, weights, mf\_params, total\_weight):  """去模糊化（重心法）"""  if total\_weight == 0:  return 0.0    weighted\_sum = 0.0  for k, v in weights.items():  if v > 0:  weighted\_sum += v \* mf\_params[k]['center']    return weighted\_sum / total\_weight    def predict(self, cod, sludge, temp):  """完整模糊推理过程"""  cod\_deg, sludge\_deg, temp\_deg = self.fuzzify(cod, sludge, temp)  do\_weights, time\_weights, total\_w = self.infer(cod\_deg, sludge\_deg, temp\_deg)    do\_delta = self.defuzzify(do\_weights, self.do\_delta\_mf, total\_w)  time\_delta = self.defuzzify(time\_weights, self.time\_delta\_mf, total\_w)    # 限制调整范围  do\_delta = np.clip(do\_delta, -1.0, 1.0)  time\_delta = np.clip(time\_delta, -2.0, 2.0)    return do\_delta, time\_delta  # 3. 模糊神经网络（FNN）模型  class FuzzyNeuralNetwork:  def \_\_init\_\_(self, fuzzy\_system, learning\_rate=0.01):  """  模糊神经网络  将模糊系统的隶属度函数参数作为可训练的权重  """  self.fuzzy = fuzzy\_system  self.lr = learning\_rate    # 初始化可训练参数（隶属度函数的中心和宽度）  # 使用模糊系统的初始参数  self.params = {  'cod': copy.deepcopy(fuzzy\_system.cod\_mf),  'sludge': copy.deepcopy(fuzzy\_system.sludge\_mf),  'temp': copy.deepcopy(fuzzy\_system.temp\_mf),  'do\_delta': copy.deepcopy(fuzzy\_system.do\_delta\_mf),  'time\_delta': copy.deepcopy(fuzzy\_system.time\_delta\_mf)  }    def membership(self, x, center, width):  """梯形隶属度函数（与模糊系统相同）"""  if x < center - width:  return 0.0  elif center - width <= x < center:  return (x - (center - width)) / width  elif center <= x < center + width:  return (center + width - x) / width  else:  return 0.0    def forward(self, cod, sludge, temp):  """前向传播：计算输出和中间变量"""  # 模糊化（使用当前参数）  cod\_degree = {k: self.membership(cod, v['center'], v['width'])  for k, v in self.params['cod'].items()}  sludge\_degree = {k: self.membership(sludge, v['center'], v['width'])  for k, v in self.params['sludge'].items()}  temp\_degree = {k: self.membership(temp, v['center'], v['width'])  for k, v in self.params['temp'].items()}    # 推理  do\_weights = {k: 0.0 for k in self.params['do\_delta'].keys()}  time\_weights = {k: 0.0 for k in self.params['time\_delta'].keys()}  total\_weight = 0.0    for (c, s, t), (do, time) in self.fuzzy.rules:  strength = min(cod\_degree[c], sludge\_degree[s], temp\_degree[t])    if strength > 0:  do\_weights[do] += strength  time\_weights[time] += strength  total\_weight += strength    # 去模糊化  do\_delta = 0.0  time\_delta = 0.0    if total\_weight > 0:  do\_delta = sum(w \* self.params['do\_delta'][k]['center']  for k, w in do\_weights.items()) / total\_weight  time\_delta = sum(w \* self.params['time\_delta'][k]['center']  for k, w in time\_weights.items()) / total\_weight    # 限制范围  do\_delta = np.clip(do\_delta, -1.0, 1.0)  time\_delta = np.clip(time\_delta, -2.0, 2.0)    # 保存中间结果用于反向传播  self.cache = {  'cod\_degree': cod\_degree,  'sludge\_degree': sludge\_degree,  'temp\_degree': temp\_degree,  'do\_weights': do\_weights,  'time\_weights': time\_weights,  'total\_weight': total\_weight  }    return do\_delta, time\_delta    def backward(self, cod, sludge, temp, do\_delta\_pred, time\_delta\_pred, do\_delta\_true, time\_delta\_true):  """反向传播：计算梯度并更新参数"""  # 计算误差  do\_error = do\_delta\_pred - do\_delta\_true  time\_error = time\_delta\_pred - time\_delta\_true    # 获取缓存的中间结果  cache = self.cache  if cache['total\_weight'] == 0:  return # 无有效推理，不更新    # 1. 更新输出隶属度函数的中心  # DO调整量  for k in self.params['do\_delta'].keys():  grad = (cache['do\_weights'][k] / cache['total\_weight']) \* do\_error  self.params['do\_delta'][k]['center'] -= self.lr \* grad    # 曝气时间调整量  for k in self.params['time\_delta'].keys():  grad = (cache['time\_weights'][k] / cache['total\_weight']) \* time\_error  self.params['time\_delta'][k]['center'] -= self.lr \* grad</doubaocanvas> |

### 13. 多目标粒子群优化（MOPSO）+ 熵权法（EWM）组合模型案例题目

**题目：电商平台绿色物流包装方案优化问题**

* **问题背景**：某电商平台年配送包裹 10 亿件，包装材料（纸箱、塑料袋、缓冲泡沫）的过度使用导致垃圾处理压力大，且运输过程中因包装破损产生的退货率达 3%。平台计划推广可循环包装，但需平衡成本与环保效益。
* **问题描述**：需从 5 种候选包装方案（如普通纸箱、可折叠循环箱、玉米淀粉缓冲材料等）中选择最优组合，目标包括：① 最小化单位包装成本（元 / 件）；② 最小化碳排放（kgCO₂/ 件）；③ 最小化破损率（≤1%）。需基于客观数据赋权，避免主观偏好影响决策。
* **数据情况**：提供各方案的成本构成（材料 + 回收 + 运输）、碳排放系数（含生产和回收环节）、不同商品类型（如电子件、衣物、食品）的破损率测试数据，以及过去 1 年的商品配送量分布。

### 13. 多目标粒子群优化（MOPSO）+ 熵权法（EWM）求解电商平台绿色物流包装方案优化代码

|  |
| --- |
| import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  import seaborn as sns  import random  import copy  from sklearn.preprocessing import MinMaxScaler  from matplotlib.patches import Patch  from matplotlib.colors import LinearSegmentedColormap  # 设置随机种子，保证结果可复现  np.random.seed(42)  random.seed(42)  # 1. 数据生成与初始化  def generate\_packaging\_data():  """  生成电商绿色物流包装方案优化问题的数据  包含5种候选包装方案的各项性能指标  """  # 1.1 候选包装方案基本信息  packaging\_schemes = pd.DataFrame({  '方案ID': ['S1', 'S2', 'S3', 'S4', 'S5'],  '方案名称': [  '普通纸箱+塑料缓冲',  '可折叠循环纸箱',  '玉米淀粉缓冲+再生纸箱',  '全生物降解包装',  '模块化共享包装'  ],  '材料类型': [  '不可再生',  '可循环',  '部分生物降解',  '全生物降解',  '可循环+模块化'  ],  '回收难度': [3, 1, 2, 2, 1], # 1-低，5-高  '适用商品类型': [  '全品类', '非生鲜', '非液体', '食品类', '3C电子'  ]  })    # 1.2 成本数据（元/件）  cost\_data = pd.DataFrame({  '方案ID': ['S1', 'S2', 'S3', 'S4', 'S5'],  '材料成本': [1.2, 5.8, 2.5, 4.2, 8.5],  '生产加工费': [0.3, 1.2, 0.5, 1.0, 1.8],  '回收处理费': [0.1, 2.0, 0.8, 0.3, 2.5],  '运输附加费': [0.2, 0.5, 0.3, 0.4, 0.6] # 因重量/体积产生的额外费用  })  # 计算单位总成本  cost\_data['单位成本'] = cost\_data.iloc[:, 1:5].sum(axis=1)    # 1.3 碳排放数据（kgCO₂/件）  carbon\_data = pd.DataFrame({  '方案ID': ['S1', 'S2', 'S3', 'S4', 'S5'],  '材料生产': [1.8, 3.2, 1.5, 2.0, 3.5],  '加工制造': [0.5, 1.0, 0.6, 0.8, 1.2],  '回收运输': [0.3, 0.8, 0.5, 0.2, 0.9],  '废弃处理': [1.2, 0.3, 0.4, 0.1, 0.2]  })  # 计算总碳排放  carbon\_data['总碳排放'] = carbon\_data.iloc[:, 1:5].sum(axis=1)    # 1.4 破损率数据（%）  # 不同商品类型的破损率  product\_types = ['电子件', '衣物', '食品', '化妆品', '书籍']  damage\_data = pd.DataFrame({  '方案ID': ['S1', 'S2', 'S3', 'S4', 'S5'],  '电子件': [2.8, 1.2, 1.5, 3.0, 0.8],  '衣物': [1.5, 0.5, 0.8, 1.2, 0.3],  '食品': [4.2, 2.5, 1.8, 1.0, 2.2],  '化妆品': [3.5, 1.8, 2.0, 2.5, 1.0],  '书籍': [1.0, 0.3, 0.5, 0.8, 0.2]  })  # 计算平均破损率  damage\_data['平均破损率'] = damage\_data.iloc[:, 1:6].mean(axis=1)    # 1.5 商品配送量分布（过去1年）  delivery\_dist = pd.DataFrame({  '商品类型': product\_types,  '配送占比': [0.25, 0.30, 0.15, 0.18, 0.12], # 各类型商品占总配送量的比例  '平均客单价': [800, 200, 150, 300, 50] # 影响破损损失  })    # 合并所有数据  merged\_data = packaging\_schemes  merged\_data = merged\_data.merge(cost\_data, on='方案ID')  merged\_data = merged\_data.merge(carbon\_data, on='方案ID')  merged\_data = merged\_data.merge(damage\_data, on='方案ID')    return {  'schemes': merged\_data,  'delivery\_dist': delivery\_dist,  'product\_types': product\_types,  'n\_schemes': 5 # 候选方案数量  }  # 2. 熵权法（EWM）计算权重  class EntropyWeightMethod:  def \_\_init\_\_(self, data):  """  初始化熵权法  data: 包含各方案和评价指标的数据框  """  self.data = data  # 提取评价指标（成本、碳排放、破损率）  self.indicators = data[['单位成本', '总碳排放', '平均破损率']].values  self.n\_schemes, self.n\_indicators = self.indicators.shape    # 指标类型：1-成本型（越小越好），2-效益型（越大越好）  self.indicator\_type = [1, 1, 1] # 三个指标均为成本型    # 标准化数据  self.standardized = self.\_standardize\_data()    # 计算熵值和权重  self.entropy = self.\_calculate\_entropy()  self.weights = self.\_calculate\_weights()    def \_standardize\_data(self):  """标准化指标数据"""  standardized = np.zeros\_like(self.indicators, dtype=np.float64)    for j in range(self.n\_indicators):  col = self.indicators[:, j]  max\_val = np.max(col)  min\_val = np.min(col)    if self.indicator\_type[j] == 1:  # 成本型指标标准化：(max - x)/(max - min)  if max\_val != min\_val:  standardized[:, j] = (max\_val - col) / (max\_val - min\_val)  else:  standardized[:, j] = 1.0  else:  # 效益型指标标准化：(x - min)/(max - min)  if max\_val != min\_val:  standardized[:, j] = (col - min\_val) / (max\_val - min\_val)  else:  standardized[:, j] = 1.0    return standardized    def \_calculate\_entropy(self):  """计算各指标的熵值"""  # 计算概率矩阵  p = np.zeros\_like(self.standardized)  for j in range(self.n\_indicators):  sum\_col = np.sum(self.standardized[:, j])  if sum\_col > 0:  p[:, j] = self.standardized[:, j] / sum\_col  else:  p[:, j] = 1.0 / self.n\_schemes    # 计算熵值  entropy = np.zeros(self.n\_indicators)  k = 1.0 / np.log(self.n\_schemes) if self.n\_schemes > 1 else 0    for j in range(self.n\_indicators):  e = -k \* np.sum(p[:, j] \* np.log(p[:, j] + 1e-10)) # 加小值避免log(0)  entropy[j] = e    return entropy    def \_calculate\_weights(self):  """计算各指标的权重"""  # 信息效用值  d = 1 - self.entropy    # 权重  if np.sum(d) > 0:  weights = d / np.sum(d)  else:  weights = np.ones(self.n\_indicators) / self.n\_indicators    return weights    def get\_weights(self):  """返回计算得到的权重"""  return self.weights    def evaluate\_scheme(self, scheme\_idx):  """计算单个方案的综合得分"""  if isinstance(scheme\_idx, int):  return np.sum(self.standardized[scheme\_idx] \* self.weights)  else:  # 如果是多个方案的组合，计算加权平均  return np.mean([np.sum(self.standardized[i] \* self.weights) for i in scheme\_idx])  # 3. 多目标粒子群优化（MOPSO）  class MOPSO:  def \_\_init\_\_(self, data, pop\_size=50, max\_iter=100, w=0.8, c1=0.5, c2=0.5):  """  初始化多目标粒子群优化算法  data: 问题数据  pop\_size: 种群大小  max\_iter: 最大迭代次数  w: 惯性权重  c1, c2: 学习因子  """  self.data = data  self.schemes = data['schemes']  self.n\_schemes = data['n\_schemes']  self.product\_types = data['product\_types']  self.delivery\_dist = data['delivery\_dist']    # 算法参数  self.pop\_size = pop\_size  self.max\_iter = max\_iter  self.w = w # 惯性权重  self.c1 = c1 # 认知系数  self.c2 = c2 # 社会系数    # 目标函数数量  self.n\_objectives = 3 # 成本、碳排放、破损率    # 初始化粒子群  self.particles = [] # 粒子位置（方案组合）  self.velocities = [] # 粒子速度  self.pbest\_pos = [] # 个体最优位置  self.pbest\_obj = [] # 个体最优目标值  self.gbest\_pos = None # 全局最优位置  self.gbest\_obj = None # 全局最优目标值    # 外部存档（存储非支配解）  self.archive = []  self.archive\_size = pop\_size    # 熵权法计算权重  self.ewm = EntropyWeightMethod(self.schemes)  self.weights = self.ewm.get\_weights()  print(f"熵权法计算的指标权重: 成本={self.weights[0]:.4f}, 碳排放={self.weights[1]:.4f}, 破损率={self.weights[2]:.4f}")    # 初始化  self.\_initialize\_particles()    # 记录优化历史  self.history = {  'gbest': [],  'archive\_size': []  }    def \_initialize\_particles(self):  """初始化粒子群"""  for \_ in range(self.pop\_size):  # 粒子位置：为每种商品类型分配一种包装方案  # 生成一个长度为商品类型数量的向量，每个元素是方案索引  pos = [random.randint(0, self.n\_schemes - 1) for \_ in self.product\_types]  self.particles.append(pos)    # 初始化速度（方案变化的概率）  vel = [np.random.uniform(-0.5, 0.5) for \_ in self.product\_types]  self.velocities.append(vel)    # 计算目标值  obj\_vals = self.\_calculate\_objectives(pos)    # 初始化个体最优  self.pbest\_pos.append(pos.copy())  self.pbest\_obj.append(obj\_vals)    # 添加到外部存档  self.\_add\_to\_archive(pos, obj\_vals)    # 初始化全局最优（从存档中选择）  self.\_update\_gbest()    def \_calculate\_objectives(self, pos):  """计算目标函数值"""  # pos: 每种商品类型对应的包装方案索引    # 1. 计算加权平均单位成本  total\_cost = 0.0  # 2. 计算加权平均碳排放  total\_carbon = 0.0  # 3. 计算加权平均破损率  total\_damage = 0.0    # 按商品类型的配送占比加权  for i, prod\_type in enumerate(self.product\_types):  scheme\_idx = pos[i]  dist\_ratio = self.delivery\_dist.loc[i, '配送占比']    # 累加各目标值  total\_cost += self.schemes.iloc[scheme\_idx]['单位成本'] \* dist\_ratio  total\_carbon += self.schemes.iloc[scheme\_idx]['总碳排放'] \* dist\_ratio  total\_damage += self.schemes.iloc[scheme\_idx][prod\_type] \* dist\_ratio    return [total\_cost, total\_carbon, total\_damage]    def \_is\_dominated(self, obj1, obj2):  """判断obj1是否被obj2支配"""  # 如果obj2在所有目标上都优于或等于obj1，且至少有一个目标严格优于  return all(o2 <= o1 for o1, o2 in zip(obj1, obj2)) and any(o2 < o1 for o1, o2 in zip(obj1, obj2))    def \_add\_to\_archive(self, pos, obj\_vals):  """将解添加到外部存档，保持存档中的解为非支配解"""  # 检查是否被存档中的解支配  dominated = False  to\_remove = []    for i, (a\_pos, a\_obj) in enumerate(self.archive):  if self.\_is\_dominated(obj\_vals, a\_obj):  # 当前解被存档中的解支配，不添加  dominated = True  break  if self.\_is\_dominated(a\_obj, obj\_vals):  # 存档中的解被当前解支配，标记为待删除  to\_remove.append(i)    if not dominated:  # 移除被支配的解（从后往前删）  for i in sorted(to\_remove, reverse=True):  del self.archive[i]    # 添加新解  self.archive.append((pos.copy(), obj\_vals.copy()))    # 如果存档过大，进行截断  if len(self.archive) > self.archive\_size:  self.\_truncate\_archive()    def \_truncate\_archive(self):  """当存档大小超过限制时，截断存档（保留分布性好的解）"""  # 计算存档中各解之间的距离  distances = []  n\_archive = len(self.archive)    for i in range(n\_archive):  dist = 0.0  for j in range(n\_archive):  if i != j:  # 计算目标空间中的欧氏距离  obj\_dist = np.sqrt(np.sum([  (self.archive[i][1][k] - self.archive[j][1][k])\*\*2  for k in range(self.n\_objectives)  ]))  dist += obj\_dist  # 平均距离  avg\_dist = dist / (n\_archive - 1) if n\_archive > 1 else 0  distances.append((i, avg\_dist))    # 按平均距离升序排序（距离小的解被移除）  distances.sort(key=lambda x: x[1])    # 移除多余的解  to\_remove = [d[0] for d in distances[:len(self.archive) - self.archive\_size]]  for i in sorted(to\_remove, reverse=True):  del self.archive[i]    def \_update\_gbest(self):  """从存档中选择全局最优解（使用轮盘赌选择）"""  if not self.archive:  return    # 计算存档中各解的适应度（基于熵权法的综合得分）  fitness = []  for pos, obj in self.archive:  # 计算该方案组合的综合得分  # 先获取每种商品类型对应的方案  scheme\_indices = list(set(pos)) # 去重  score = self.ewm.evaluate\_scheme(scheme\_indices)  fitness.append(score)    # 轮盘赌选择  total\_fitness = sum(fitness)  if total\_fitness == 0:  # 所有适应度相同，随机选择  self.gbest\_pos, self.gbest\_obj = random.choice(self.archive)  else:  probabilities = [f / total</doubaocanvas> |

### 14. 遗传算法（GA）+ 约束规划（CP）组合模型案例题目

**题目：航空公司机组人员排班优化问题**

* **问题背景**：某航空公司有 100 条国内航线，需为每日 300 个航班分配机组人员（飞行员 + 乘务员）。机组人员需满足严格的约束：如单日飞行时间≤8 小时、连续休息≥10 小时、每月飞行时间≤100 小时，且特定机型需匹配持证飞行员。当前排班依赖人工，常出现合规性冲突（月冲突率 12%）。
* **问题描述**：需优化机组人员与航班的匹配方案，目标包括：① 满足所有法规约束（冲突率 = 0）；② 最小化机组人员出差住宿成本；③ 平衡各机组的月飞行时长（差异≤5 小时）。约束条件复杂且相互关联（如休息时间受前一日航班结束时间影响）。
* **数据情况**：提供航班时刻表（起飞 / 降落时间、机型、航线）、机组人员信息（资质、居住地、月累计飞行时间）、不同城市的住宿成本、法规约束细则（如飞行时间限制条款）。

### 14. 遗传算法（GA）+ 约束规划（CP）求解航空公司机组人员排班优化代码

|  |
| --- |
| import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  import seaborn as sns  import random  import copy  from datetime import datetime, timedelta  import itertools  from collections import defaultdict  # 设置随机种子，保证结果可复现  np.random.seed(42)  random.seed(42)  # 1. 数据生成与初始化  def generate\_airline\_data(n\_flights=300, n\_crews=150, n\_days=30):  """  生成航空公司机组人员排班问题的数据  n\_flights: 每日航班数量  n\_crews: 机组人员总数  n\_days: 优化周期（天数）  """  # 1.1 机场与城市信息  cities = [  "北京", "上海", "广州", "深圳", "成都",  "杭州", "重庆", "西安", "武汉", "南京",  "沈阳", "大连", "青岛", "厦门", "昆明"  ]  n\_cities = len(cities)    # 城市间距离（km）和住宿成本（元/晚）  city\_data = pd.DataFrame({  "城市": cities,  "住宿成本": np.random.randint(300, 801, size=n\_cities), # 300-800元/晚  "机场代码": [f"CK{i+1:02d}" for i in range(n\_cities)]  })    # 1.2 机型信息  aircraft\_types = pd.DataFrame({  "机型": ["A320", "B737", "A330", "B777"],  "所需飞行员": [2, 2, 3, 4],  "所需乘务员": [4, 4, 8, 10],  "资质要求": ["C1", "C2", "C3", "C4"]  })    # 1.3 航班数据  flights = []  flight\_ids = [f"FL{i+1:03d}" for i in range(n\_flights)]    # 生成每日航班计划（简化为一天的航班，可扩展到n\_days天）  for flight\_id in flight\_ids:  # 随机选择起飞机场和目的机场  dep\_city\_idx, arr\_city\_idx = random.sample(range(n\_cities), 2)  dep\_city = cities[dep\_city\_idx]  arr\_city = cities[arr\_city\_idx]    # 飞行时间（小时）  distance = np.random.randint(500, 2501) # 500-2500km  flight\_time = distance / 800 # 平均时速800km/h    # 起飞时间（一天中的随机时间）  dep\_hour = random.randint(6, 20) # 6:00-20:00之间起飞  dep\_minute = random.choice([0, 15, 30, 45])  dep\_time = datetime(2023, 1, 1, dep\_hour, dep\_minute)    # 到达时间  arr\_time = dep\_time + timedelta(hours=flight\_time)    # 选择机型  aircraft = random.choice(aircraft\_types["机型"].values)  ac\_info = aircraft\_types[aircraft\_types["机型"] == aircraft].iloc[0]    flights.append({  "航班ID": flight\_id,  "起飞城市": dep\_city,  "到达城市": arr\_city,  "起飞时间": dep\_time,  "到达时间": arr\_time,  "飞行时长": round(flight\_time, 2),  "机型": aircraft,  "所需飞行员": ac\_info["所需飞行员"],  "所需乘务员": ac\_info["所需乘务员"],  "资质要求": ac\_info["资质要求"],  "住宿成本": city\_data[city\_data["城市"] == arr\_city]["住宿成本"].values[0]  })    flights\_df = pd.DataFrame(flights)    # 1.4 机组人员数据  crew\_types = ["飞行员", "乘务员"]  crews = []    for crew\_id in range(n\_crews):  # 随机选择人员类型  crew\_type = random.choice(crew\_types)    # 飞行员有资质要求，乘务员无  if crew\_type == "飞行员":  # 随机拥有1-2种资质  qualifications = random.sample(aircraft\_types["资质要求"].unique().tolist(),  random.randint(1, 2))  else:  qualifications = []    # 居住地  base\_city = random.choice(cities)    # 月累计飞行时间（优化开始前）  monthly\_hours = random.uniform(40, 80) # 40-80小时    # 日累计飞行时间（初始为0）  daily\_hours = 0.0    crews.append({  "人员ID": f"CR{crew\_id+1:03d}",  "类型": crew\_type,  "资质": qualifications,  "居住地": base\_city,  "月累计飞行时间": round(monthly\_hours, 2),  "日累计飞行时间": daily\_hours,  "最后休息开始时间": datetime(2023, 1, 1, 0, 0), # 初始休息时间  "连续休息时长": 12.0 # 初始有12小时休息  })    crews\_df = pd.DataFrame(crews)    # 1.5 法规约束参数  regulations = {  "单日最大飞行时间": 8.0, # 小时  "连续最小休息时间": 10.0, # 小时  "每月最大飞行时间": 100.0, # 小时  "最大连续工作天数": 6, # 天  "最小连续休息天数": 1, # 天  "同机组最大连续合作天数": 5 # 天  }    return {  "航班数据": flights\_df,  "机组人员数据": crews\_df,  "城市数据": city\_data,  "机型数据": aircraft\_types,  "法规约束": regulations,  "优化周期": n\_days,  "城市列表": cities  }  # 2. 约束规划（CP）检查与修复  class CrewSchedulingCP:  def \_\_init\_\_(self, data):  """  初始化约束规划模块  data: 包含航班、机组人员和约束规则的数据  """  self.data = data  self.regulations = data["法规约束"]  self.flights = data["航班数据"]  self.crews = data["机组人员数据"].copy()    # 记录机组人员的排班历史  self.crew\_history = defaultdict(list) # crew\_id -> 排班记录  self.conflict\_stats = defaultdict(int) # 冲突类型统计    def check\_constraints(self, assignment, day):  """  检查单个航班的机组分配是否满足所有约束  assignment: {航班ID: {飞行员列表, 乘务员列表}}  day: 当前日期（用于计算月度累计）  返回值: 是否满足所有约束，冲突信息  """  conflicts = []  all\_valid = True    # 1. 检查每个航班的人员配置是否满足机型要求  for flight\_id, crew\_assign in assignment.items():  flight\_info = self.flights[self.flights["航班ID"] == flight\_id].iloc[0]  required\_pilots = flight\_info["所需飞行员"]  required\_cabin = flight\_info["所需乘务员"]  required\_qual = flight\_info["资质要求"]    # 人数检查  if len(crew\_assign["飞行员"]) != required\_pilots:  conflicts.append({  "类型": "人员数量不足",  "航班ID": flight\_id,  "详情": f"需要{required\_pilots}名飞行员，实际分配{len(crew\_assign['飞行员'])}名"  })  all\_valid = False    if len(crew\_assign["乘务员"]) != required\_cabin:  conflicts.append({  "类型": "人员数量不足",  "航班ID": flight\_id,  "详情": f"需要{required\_cabin}名乘务员，实际分配{len(crew\_assign['乘务员'])}名"  })  all\_valid = False    # 资质检查（飞行员）  for pilot\_id in crew\_assign["飞行员"]:  pilot\_info = self.crews[self.crews["人员ID"] == pilot\_id].iloc[0]  if required\_qual not in pilot\_info["资质"]:  conflicts.append({  "类型": "资质不匹配",  "航班ID": flight\_id,  "人员ID": pilot\_id,  "详情": f"飞行员无{required\_qual}资质，无法执行该机型航班"  })  all\_valid = False    # 2. 检查每个机组人员的约束  all\_assigned\_crews = []  for crew\_assign in assignment.values():  all\_assigned\_crews.extend(crew\_assign["飞行员"])  all\_assigned\_crews.extend(crew\_assign["乘务员"])    # 去重  all\_assigned\_crews = list(set(all\_assigned\_crews))    for crew\_id in all\_assigned\_crews:  crew\_info = self.crews[self.crews["人员ID"] == crew\_id].iloc[0]  crew\_history = self.crew\_history[crew\_id]    # 找到该人员当天分配的所有航班  assigned\_flights = []  for flight\_id, crew\_assign in assignment.items():  if crew\_id in crew\_assign["飞行员"] or crew\_id in crew\_assign["乘务员"]:  assigned\_flights.append(flight\_id)    # 计算当天总飞行时间  total\_flight\_time = 0.0  flight\_times = []  for flight\_id in assigned\_flights:  flight\_info = self.flights[self.flights["航班ID"] == flight\_id].iloc[0]  total\_flight\_time += flight\_info["飞行时长"]  flight\_times.append((flight\_info["起飞时间"], flight\_info["到达时间"]))    # 2.1 单日飞行时间约束  if total\_flight\_time > self.regulations["单日最大飞行时间"]:  conflicts.append({  "类型": "单日飞行超时",  "人员ID": crew\_id,  "详情": f"当日飞行{total\_flight\_time:.2f}小时，超过最大限制{self.regulations['单日最大飞行时间']}小时"  })  all\_valid = False  self.conflict\_stats["单日飞行超时"] += 1    # 2.2 休息时间约束（与前一天最后航班比较）  if crew\_history:  last\_day = max([h["日期"] for h in crew\_history])  last\_flights = [h for h in crew\_history if h["日期"] == last\_day]    if last\_flights:  # 前一天最后到达时间  last\_arrival = max([f["到达时间"] for h in last\_flights for f in h["航班"]])  # 今天最早起飞时间  first\_departure = min([ft[0] for ft in flight\_times])    # 计算休息时长  rest\_duration = (first\_departure - last\_arrival).total\_seconds() / 3600    if rest\_duration < self.regulations["连续最小休息时间"]:  conflicts.append({  "类型": "休息时间不足",  "人员ID": crew\_id,  "详情": f"仅休息{rest\_duration:.2f}小时，低于最小要求{self.regulations['连续最小休息时间']}小时"  })  all\_valid = False  self.conflict\_stats["休息时间不足"] += 1    # 2.3 月度飞行时间约束  current\_monthly = crew\_info["月累计飞行时间"] + total\_flight\_time  if current\_monthly > self.regulations["每月最大飞行时间"]:  conflicts.append({  "类型": "月度飞行超时",  "人员ID": crew\_id,  "详情": f"月度累计将达{current\_monthly:.2f}小时，超过最大限制{self.regulations['每月最大飞行时间']}小时"  })  all\_valid = False  self.conflict\_stats["月度飞行超时"] += 1    return all\_valid, conflicts    def repair\_solution(self, assignment, day):  """  修复不满足约束的排班方案  assignment: 初始排班方案  day: 当前日期  返回值: 修复后的排班方案  """  # 检查当前方案  valid, conflicts = self.check\_constraints(assignment, day)  if valid:  return assignment    # 统计冲突人员  conflict\_crews = set()  for c in conflicts:  if "人员ID" in c:  conflict\_crews.add(c["人员ID"])    # 收集可用的替代人员  all\_crews = self.crews["人员ID"].tolist()  available\_crews = [c for c in all\_crews if c not in conflict\_crews]    # 按冲突类型进行修复  for conflict in conflicts:  if conflict["类型"] == "资质不匹配":  # 替换为有相应资质的飞行员  flight\_id = conflict["航班ID"]  crew\_id = conflict["人员ID"]  flight\_info = self.flights[self.flights["航班ID"] == flight\_id].iloc[0]  required\_qual = flight\_info["资质要求"]    # 找到该航班的分配  for fa in assignment.values():  if flight\_id in fa:  # 找到该飞行员在列表中的位置  if crew\_id in fa["飞行员"]:  idx = fa["飞行员"].index(crew\_id)    # 寻找替代飞行员  eligible\_pilots = []  for c in available\_crews:  c\_info = self.crews[self.crews["人员ID"] == c].iloc[0]  if c\_info["类型"] == "飞行员" and required\_qual in c\_info["资质"]:  eligible\_pilots.append(c)    if eligible\_pilots:  # 选择一个替代  replacement = random.choice(eligible\_pilots)  fa["飞行员"][idx] = replacement  available\_crews.remove(replacement)  conflict\_crews.add(crew\_id)  available\_crews.append(crew\_id)    # 再次检查修复后的方案  valid, \_ = self.check\_constraints(assignment, day)  if valid:  return assignment    # 如果仍有冲突，进行更深度的修复（随机替换冲突人员）  for crew\_id in conflict\_crews:  # 找到该人员分配的所有航班  flights\_to\_reassign = []  for flight\_id, fa in assignment.items():  if crew\_id in fa["飞行员"]:  flights\_to\_reassign.append((flight\_id, "飞行员", fa["飞行员"].index(crew\_id)))  elif crew\_id in fa["乘务员"]:  flights\_to\_reassign.append((flight\_id, "乘务员", fa["乘务员"].index(crew\_id)))    # 为每个航班重新分配人员  for flight\_id, crew\_type, idx in flights\_to\_reassign:  flight\_info = self.flights[self.flights["航班ID"] == flight\_id].iloc[0]    # 寻找合适的替代人员  eligible = []  for c in available\_crews:  c\_info = self.crews[self.crews["人员ID"] == c].iloc[0]  if c\_info["类型"] == crew\_type:  # 飞行员需要检查资质  if crew\_type == "飞行员":  if flight\_info["资质要求"] in c\_info["资质"]:  eligible.append(c)  else:  eligible.append(c)    if eligible:  replacement = random.choice(eligible)  # 更新分配  assignment[flight\_id][crew\_type][idx] = replacement  available\_crews.remove(replacement)  available\_crews.append(crew\_id)    return assignment    def update\_crew\_history(self, assignment, day):  """更新机组人员的排班历史"""  for flight\_id, crew\_assign in assignment.items():  flight\_info = self.flights[self.flights["航班ID"] == flight\_id].iloc[0]  flight\_data = {  "航班ID": flight\_id,  "起飞时间": flight\_info["起飞时间"],  "到达时间": flight\_info["到达时间"],  "飞行时长": flight\_info["飞行时长"],  "到达城市": flight\_info["到达城市"]  }    # 更新飞行员历史  for pilot\_id in crew\_assign["飞行员"]:  self.crew\_history[pilot\_id].append({  "日期": day,  "航班": [flight\_data]  })  # 更新累计飞行时间  self.crews.loc[self.crews["人员ID"] ==</doubaocanvas> |

### 15. 模拟退火（SA）+ 神经网络（NN）组合模型案例题目

**题目：智慧农业温室环境调控优化问题**

**・问题背景：**某现代化温室基地种植高端果蔬，当前依赖传统 PID 控制或人工经验设定环境参数，存在能耗过高（占生产成本 45%）、调控滞后（响应延迟超 30 分钟）、品质波动（差异率超 15%）及多目标冲突等问题。基地已部署传感器网络和智能执行设备，具备精准调控基础。

**・问题描述：**需优化温室环境调控参数，目标包括：① 最大化作物周生长量（≥50g / 株）；② 最大化果实品质评分（≥85 分）；③ 最小化单位产量能耗（≤8kWh/kg）。需遵循温度、湿度、CO₂浓度等约束，每 30 分钟调整一次参数。

**・数据情况：**提供近 2 年历史数据，包括每 5 分钟采集的温室内外环境数据、每周的作物生长指标、收获期的品质数据，以及分设备记录的能耗数据，数据含一定噪声和缺失值。

### 15. 模拟退火（SA）+ 神经网络（NN）求解智慧农业温室环境调控优化代码

|  |
| --- |
| import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  import seaborn as sns  import random  import copy  from datetime import datetime, timedelta  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import MinMaxScaler  from sklearn.metrics import mean\_squared\_error, r2\_score  import torch  import torch.nn as nn  import torch.optim as optim  from torch.utils.data import Dataset, DataLoader  # 设置随机种子，保证结果可复现  np.random.seed(42)  random.seed(42)  torch.manual\_seed(42)  # 1. 数据生成与预处理  def generate\_greenhouse\_data(n\_days=730, n\_samples\_day=288):  """  生成温室环境调控问题数据（模拟2年数据）  n\_days: 天数  n\_samples\_day: 每天的采样次数（5分钟一次，288次/天）  """  # 时间序列  time\_stamps = []  current\_time = datetime(2022, 1, 1, 0, 0)  for \_ in range(n\_days \* n\_samples\_day):  time\_stamps.append(current\_time)  current\_time += timedelta(minutes=5)    # 基础数据框架  data = pd.DataFrame({  '时间': time\_stamps,  '季节': [t.month % 12 for t in time\_stamps], # 1-12月  '小时': [t.hour for t in time\_stamps]  })    # 1.1 外部环境数据  # 室外温度（受季节和时间影响）  base\_temp = 15 + 10 \* np.sin(data['季节'] / 12 \* 2 \* np.pi)  hourly\_temp = 5 \* np.sin((data['小时'] - 12) / 24 \* 2 \* np.pi)  data['室外温度(℃)'] = base\_temp + hourly\_temp + np.random.normal(0, 1.5, len(data))  data['室外温度(℃)'] = np.clip(data['室外温度(℃)'], -5, 35)    # 室外光照（白天有光照）  daylight = np.where((data['小时'] >= 6) & (data['小时'] <= 18), 1, 0)  data['室外光照(lux)'] = daylight \* (50000 + 30000 \* np.sin((data['小时'] - 12) / 12 \* np.pi))  data['室外光照(lux)'] += np.random.normal(0, 5000, len(data))  data['室外光照(lux)'] = np.clip(data['室外光照(lux)'], 0, 100000)    # 1.2 温室调控参数（历史设定值）  # 日间温度设定（20-28℃）  day\_temp = 24 + 2 \* np.sin((data['小时'] - 12) / 12 \* np.pi)  # 夜间温度设定（15-18℃）  night\_temp = 16 + np.sin((data['小时'] - 22) / 12 \* np.pi)  # 昼夜区分  is\_day = np.where((data['小时'] >= 6) & (data['小时'] <= 18), 1, 0)  data['温度设定(℃)'] = is\_day \* day\_temp + (1 - is\_day) \* night\_temp  data['温度设定(℃)'] += np.random.normal(0, 0.5, len(data))  data['温度设定(℃)'] = np.clip(data['温度设定(℃)'], 15, 28)    # 湿度设定（60%-80%）  data['湿度设定(%)'] = 70 + 5 \* np.sin((data['小时'] - 8) / 16 \* np.pi)  data['湿度设定(%)'] += np.random.normal(0, 2, len(data))  data['湿度设定(%)'] = np.clip(data['湿度设定(%)'], 60, 80)    # CO₂浓度设定（400-1200ppm）  data['CO2设定(ppm)'] = 800 + 300 \* is\_day  data['CO2设定(ppm)'] += np.random.normal(0, 50, len(data))  data['CO2设定(ppm)'] = np.clip(data['CO2设定(ppm)'], 400, 1200)    # 补光时长（小时/天）  light\_needed = np.where(data['室外光照(lux)'] < 30000, 1, 0)  data['补光设定(小时)'] = light\_needed \* (3 + 2 \* np.sin((data['小时'] - 10) / 8 \* np.pi))  data['补光设定(小时)'] = np.clip(data['补光设定(小时)'], 0, 8)    # 1.3 实际环境数据（受设定值和外部环境影响）  data['室内温度(℃)'] = data['温度设定(℃)'] + 0.3 \* (data['室外温度(℃)'] - data['温度设定(℃)'])  data['室内温度(℃)'] += np.random.normal(0, 0.3, len(data))    data['室内湿度(%)'] = data['湿度设定(%)'] + 0.2 \* np.random.normal(0, 5, len(data))  data['室内湿度(%)'] = np.clip(data['室内湿度(%)'], 50, 90)    data['室内CO2(ppm)'] = data['CO2设定(ppm)'] + np.random.normal(0, 30, len(data))  data['室内CO2(ppm)'] = np.clip(data['室内CO2(ppm)'], 350, 1300)    # 1.4 作物生长与品质数据（每周记录，这里按比例扩展）  # 生长阶段（0-2：苗期、花期、结果期）  growth\_phase = np.floor((np.arange(len(data)) / (n\_samples\_day \* 30)) % 3).astype(int)  data['生长阶段'] = growth\_phase    # 周生长量（每30分钟的累积值）  temp\_factor = np.exp(-0.1 \* np.abs(data['室内温度(℃)'] - 25))  co2\_factor = data['室内CO2(ppm)'] / 1000  light\_factor = np.minimum(data['室外光照(lux)'] / 50000 + data['补光设定(小时)'] / 8, 1)  data['生长量(g/株)'] = 0.001 \* temp\_factor \* co2\_factor \* light\_factor  data['生长量(g/株)'] += np.random.normal(0, 0.0002, len(data))    # 品质评分（综合值）  data['品质评分'] = 70 + 15 \* temp\_factor + 5 \* co2\_factor + np.random.normal(0, 2, len(data))  data['品质评分'] = np.clip(data['品质评分'], 50, 100)    # 1.5 能耗数据（kWh）  # 加温能耗（温度低于设定值时）  heating = np.where(data['室外温度(℃)'] < data['温度设定(℃)'],  (data['温度设定(℃)'] - data['室外温度(℃)']) \* 0.5, 0)  # 降温能耗（温度高于设定值时）  cooling = np.where(data['室外温度(℃)'] > data['温度设定(℃)'],  (data['室外温度(℃)'] - data['温度设定(℃)']) \* 0.3, 0)  # 补光能耗  lighting = data['补光设定(小时)'] / 24 \* 1.2  # CO2生成能耗  co2\_energy = (data['CO2设定(ppm)'] - 400) / 800 \* 0.8    data['能耗(kWh)'] = heating + cooling + lighting + co2\_energy + np.random.normal(0, 0.1, len(data))  data['能耗(kWh)'] = np.clip(data['能耗(kWh)'], 0, 5)    # 按30分钟聚合数据（调控周期）  data['调控周期'] = (data.index // 6).astype(int) # 每6个5分钟为30分钟  agg\_data = data.groupby('调控周期').agg({  '时间': 'first',  '季节': 'first',  '室外温度(℃)': 'mean',  '室外光照(lux)': 'mean',  '温度设定(℃)': 'mean',  '湿度设定(%)': 'mean',  'CO2设定(ppm)': 'mean',  '补光设定(小时)': 'sum', # 累计补光时间  '生长阶段': 'first',  '生长量(g/株)': 'sum', # 累计生长量  '品质评分': 'mean',  '能耗(kWh)': 'sum' # 累计能耗  }).reset\_index(drop=True)    # 计算单位产量能耗（每克生长量的能耗）  agg\_data['单位产量能耗(kWh/g)'] = agg\_data['能耗(kWh)'] / (agg\_data['生长量(g/株)'] + 0.001)    return agg\_data  # 2. 神经网络模型（预测生长、品质和能耗）  class GreenhouseNN(nn.Module):  def \_\_init\_\_(self, input\_size=8, hidden\_sizes=[64, 32, 16], output\_size=3):  """  温室环境预测神经网络  input\_size: 输入特征数量  hidden\_sizes: 隐藏层大小列表  output\_size: 输出特征数量（生长量、品质、能耗）  """  super(GreenhouseNN, self).\_\_init\_\_()    # 构建网络层  layers = []  prev\_size = input\_size  for size in hidden\_sizes:  layers.append(nn.Linear(prev\_size, size))  layers.append(nn.ReLU())  prev\_size = size  layers.append(nn.Linear(prev\_size, output\_size))    self.model = nn.Sequential(\*layers)    # 初始化权重  self.\_initialize\_weights()    def \_initialize\_weights(self):  """初始化网络权重"""  for m in self.model.modules():  if isinstance(m, nn.Linear):  nn.init.kaiming\_normal\_(m.weight, mode='fan\_in', nonlinearity='relu')  if m.bias is not None:  nn.init.constant\_(m.bias, 0)    def forward(self, x):  """前向传播"""  return self.model(x)  # 数据加载器  class GreenhouseDataset(Dataset):  def \_\_init\_\_(self, data, scaler\_X=None, scaler\_y=None, is\_train=True):  """  温室数据数据集  data: 原始数据  scaler\_X: 输入特征标准化器  scaler\_y: 输出特征标准化器  is\_train: 是否为训练集（决定是否拟合标准化器）  """  # 选择特征  self.features = [  '季节', '室外温度(℃)', '室外光照(lux)',  '温度设定(℃)', '湿度设定(%)', 'CO2设定(ppm)',  '补光设定(小时)', '生长阶段'  ]  self.targets = [  '生长量(g/株)', '品质评分', '能耗(kWh)'  ]    # 提取特征和目标  X = data[self.features].values  y = data[self.targets].values    # 标准化  if is\_train:  self.scaler\_X = MinMaxScaler()  self.scaler\_y = MinMaxScaler()  self.X = self.scaler\_X.fit\_transform(X)  self.y = self.scaler\_y.fit\_transform(y)  else:  self.scaler\_X = scaler\_X  self.scaler\_y = scaler\_y  self.X = self.scaler\_X.transform(X)  self.y = self.scaler\_y.transform(y)    # 转换为张量  self.X = torch.FloatTensor(self.X)  self.y = torch.FloatTensor(self.y)    def \_\_len\_\_(self):  return len(self.X)    def \_\_getitem\_\_(self, idx):  return self.X[idx], self.y[idx]  # 训练神经网络  def train\_nn\_model(data, epochs=100, batch\_size=64, lr=0.001):  """训练神经网络模型"""  # 划分训练集和验证集  train\_data, val\_data = train\_test\_split(data, test\_size=0.2, random\_state=42)    # 创建数据集  train\_dataset = GreenhouseDataset(train\_data, is\_train=True)  val\_dataset = GreenhouseDataset(val\_data,  scaler\_X=train\_dataset.scaler\_X,  scaler\_y=train\_dataset.scaler\_y,  is\_train=False)    # 创建数据加载器  train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)  val\_loader = DataLoader(val\_dataset, batch\_size=batch\_size, shuffle=False)    # 初始化模型、损失函数和优化器  model = GreenhouseNN()  criterion = nn.MSELoss()  optimizer = optim.Adam(model.parameters(), lr=lr)    # 记录训练过程  train\_losses = []  val\_losses = []    # 训练模型  for epoch in range(epochs):  model.train()  train\_loss = 0.0    for X\_batch, y\_batch in train\_loader:  # 前向传播  outputs = model(X\_batch)  loss = criterion(outputs, y\_batch)    # 反向传播和优化  optimizer.zero\_grad()  loss.backward()  optimizer.step()    train\_loss += loss.item() \* X\_batch.size(0)    # 计算平均训练损失  train\_loss /= len(train\_loader.dataset)  train\_losses.append(train\_loss)    # 在验证集上评估  model.eval()  val\_loss = 0.0  with torch.no\_grad():  for X\_batch, y\_batch in val\_loader:  outputs = model(X\_batch)  loss = criterion(outputs, y\_batch)  val\_loss += loss.item() \* X\_batch.size(0)    val\_loss /= len(val\_loader.dataset)  val\_losses.append(val\_loss)    # 每10个epoch打印一次  if (epoch + 1) % 10 == 0:  print(f'Epoch [{epoch+1}/{epochs}], 训练损失: {train\_loss:.6f}, 验证损失: {val\_loss:.6f}')    # 绘制损失曲线  plt.figure(figsize=(10, 6))  plt.plot(train\_losses, label='训练损失')  plt.plot(val\_losses, label='验证损失')  plt.title('模型训练损失曲线')  plt.xlabel('Epoch')  plt.ylabel('MSE损失')  plt.legend()  plt.grid(True)  plt.show()    return model, train\_dataset.scaler\_X, train\_dataset.scaler\_y, train\_dataset.features  # 3. 模拟退火算法（优化调控参数）  class SimulatedAnnealing:  def \_\_init\_\_(self, model, scaler\_X, scaler\_y, features, initial\_temp=100,  cooling\_rate=0.95, max\_iter=1000, final\_temp=1e-3):  """  模拟退火算法优化温室调控参数  model: 训练好的神经网络模型  scaler\_X: 输入标准化器  scaler\_y: 输出标准化器  features: 特征名称列表  initial\_temp: 初始温度  cooling\_rate: 降温速率  max\_iter: 最大迭代次数  final\_temp: 终止温度  """  self.model = model  self.scaler\_X = scaler\_X  self.scaler\_y = scaler\_y  self.features = features    # SA参数  self.T = initial\_temp  self.cooling\_rate = cooling\_rate  self.max\_iter = max\_iter  self.final\_temp = final\_temp    # 调控参数的约束范围  self.param\_ranges = {  '温度设定(℃)': (15, 28),  '湿度设定(%)': (60, 80),  'CO2设定(ppm)': (400, 1200),  '补光设定(小时)': (0</doubaocanvas> |