### 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> |