2026美赛 C 题 问题思路分析

这道题其实非常适合用**数据挖掘 + 逆向推断**的方法来做。如果你不熟悉博弈论、社会选择理论这些经济学模型，硬套反而容易露馅。用数据挖掘方法的话，核心逻辑就变成了：**基于历史淘汰结果的模式识别与投票逆推**。

背景知识： DWTS 投票机制

在动手之前，必须先搞清楚节目的投票规则。根据 Entertainment Weekly 的报道[1]和节目官方资料[2]：

评委评分：每周每对组合（明星+舞伴）表演后，由3-4位评委打分，每人满分10分。

观众投票：观众可以通过电话、短信、网站等方式投票。投票数据完全保密，从不公开！

淘汰规则：第1-2季用的是排名法（Ordinal Ranking），第3季之后改用百分比法（Percent Method）[2]。

方法	公式	说明
排名法	总排名 = 评委排名 + 投票排名	排名数字越小越好， 相加后最大的被淘汰
百分比法	总比例 = 50%×评委占比 + 50% ×投票占比	占比最低的被淘汰

关键争议：虽然官方说是50/50，但很多观众认为"观众投票权重更大"[3]。这就是问题二要分析的核心。

问题一 ：估计观众投票

问题抽象

问题一本质上是一个逆向推断问题（Inverse Problem）。

已知：评委评分、每周淘汰结果
未知：观众投票数/比例

用人话说就是：我知道某人被淘汰了，我知道他评委分是多少，我反推他观众投票大概是多少。

> 重要提醒：题目**没有指定**使用哪种投票合成方法！
>
> 历史上节目用过两种方法：**排名法（S1-S2）和百分比法（S3起）**。
>
> 所以问题一需要**分别用两种方法建模**，得到两组投票估计！

为什么说这是"逆向问题"？

正向问题是： 已知投票 → 计算谁被淘汰。这很简单，套公式就行。

逆向问题是： 已知谁被淘汰 → 反推投票。这就难了，因为：

1. **解不唯一**： 可能有无穷多组投票都能导致同一个淘汰结果
2. **信息损失**：淘汰结果只告诉你"谁最差"，不告诉你其他人之间的差距
3. **噪声敏感**：评委分的小误差可能导致投票估计的大偏差

这类问题在科学领域很常见，比如：

- 地震波数据 → 反推地下结构（地球物理）
- CT扫描图像 → 反推人体组织（ 医学成像）
- 观测轨迹 → 反推初始条件（天体力学）

为什么选择这个视角？

很多人可能会想用回归/机器学习来预测投票，但那需要**有标签的训练数据**——我恰恰没有！投票数是保密的。

逆向推断的好处是：**它不需要投票的真实值，只需要淘汰结果作为约束**。这正好匹配我的数据情况。

建模思路

如果你熟悉运筹学，这就是一个**约束满足问题**（Constraint Satisfaction Problem）

> 核心策略：针对排名法和百分比法，**分别构建两个优化模型**

为什么要分别建模？

1. 两种方法的约束条件不同：
   - 排名法：被淘汰者的（评委排名 + 投票排名）必须最大

- 百分比法：被淘汰者的（50%评委占比 + 50%投票占比）必须最低

2. 估计出的投票会不同： 同样的淘汰结果，两种方法逆推出的投票分布可能差异很大
3. 这正是问题二需要的输入： 问题二要对比两种方法，前提是问题一给出两组估计两种方法的约束条件：

方法	综合得分公式	淘汰条件
排名法	Score_j = Rank_judge(j) + Rank_vote(j)	最大的被淘汰
百分比法	Score_j = 0.5 × Pct_judge(j) + 0.5 × V_j	最小的被淘汰

目标函数（在满足约束的前提下优化）：

因为解不唯一，我需要一个先验假设来挑选"最合理"的解。

最自然的假设是：观众倾向于给高分选手投票（虽然不完全一致，但大体正相关）。

所以目标函数设为：min sum((V_j - V_hat_j)^2)，其中V_hat_j是根据评委分预测的投票比例。这样做的好处：

1. 保证估计的投票一定能复现实际淘汰结果
2. 在多个可能解中选择"最不意外"的那个
3. 可以用成熟的凸优化求解器（cvxpy、ECOS等）高效求解
4. 两种方法分别建模，为问题二提供对比基础

用 Python + cvxpy 实现约束优化（两种方法）：

```
import cvxpy as cp
import numpy as np


def estimate_votes_single_week(judge_scores, eliminated_idx, method='percent'):
    """
    逆推单周观众投票
    judge_scores: 各选手评委平均分 (list)
    eliminated_idx: 被淘汰选手的索引
    method: 'rank' 或 'percent'  ← 需要分别调用!
    """
    n = len(judge_scores)
    V = cp.Variable(n, nonneg=True)  # 投票比例，非负

    constraints = [cp.sum(V) == 1]  # 投票比例加起来等于1

    if method == 'percent':
        # ===== 百分比法 =====
        # 总得分 = 0.5*评委占比 + 0.5*投票占比，最低的被淘汰
        judge_total = sum(judge_scores)
        judge_pct = [s / judge_total for s in judge_scores]

        for j in range(n):
            if j != eliminated_idx:
                constraints.append(
                    0.5 * judge_pct[eliminated_idx] + 0.5 * V[eliminated_idx]
                    <= 0.5 * judge_pct[j] + 0.5 * V[j] - 0.001 )

    elif method == 'rank':
        # ===== 排名法 =====
        # 这里用连续松弛近似排名（因为排名是离散的）
        # 约束：被淘汰者的投票必须足够低，使其总排名最差
        # 简化处理：要求被淘汰者投票比例低于其他所有人
        for j in range(n):
            if j != eliminated_idx:
                # 被淘汰者投票比例必须比其他人低
                constraints.append(V[eliminated_idx] <= V[j] - 0.001)


    # 目标函数：让投票尽量"合理"（与评委分正相关）
    judge_normalized = np.array(judge_scores) / sum(judge_scores)
    objective = cp.Minimize(cp.sum_squares(V - judge_normalized))
```

```
    prob = cp.Problem(objective, constraints)
    prob.solve(solver=cp.ECOS)

    if prob.status == 'optimal':
        return V.value else:
        return None

# ===== 分别用两种方法估计 =====
judge_scores = [8.5, 7.2, 9.0, 6.8, 7.5] eliminated_idx = 3  # 第4个人被淘汰了

# 方法1: 百分比法
votes_percent = estimate_votes_single_week(judge_scores, eliminated_idx, method='percent') print("百分比法估计的投票：", votes_percent)

# 方法2: 排名法
votes_rank = estimate_votes_single_week(judge_scores, eliminated_idx, method='rank') print("排名法估计的投票：", votes_rank)
```

一致性验证

模型跑出来之后，怎么验证对不对？

思路：用估计出的投票，按规则重新计算淘汰结果，看和实际结果是否一致。

```
def calculate_consistency(estimated_votes_all_weeks, actual_eliminations,
judge_scores_all_weeks)
    """计算模型一致性"""
    correct = 0
    total = len(actual_eliminations)

    for week_idx in range(total):
        votes = estimated_votes_all_weeks[week_idx]
        judges = judge_scores_all_weeks[week_idx]

        # 用投票和评委分计算谁应该被淘汰
        predicted_elim = predict_elimination(votes, judges)

        if predicted_elim == actual_eliminations[week_idx]:
            correct += 1

    return correct / total  # 准确率

# 理想情况下应该是 100%
```

不确定性度量

观众投票的估计不是唯一的！可能有很多组 I 都满足约束。

方法：用 Monte Carlo 多次采样，计算置信区间。

```
import numpy as np

def estimate_uncertainty(judge_scores, eliminated_idx, n_samples=1000):
    """Bootstrap估计不确定性"""
    n = len(judge_scores)
    vote_samples = []

    for _ in range(n_samples):
        # 在评委分上加一点噪声
        noisy_scores = [s + np.random.normal(0, 0.1) for s in
        judge_scores] votes = estimate_votes_single_week(noisy_scores,
        eliminated_idx)
        if votes is not None:
            vote_samples.append(votes)

    vote_samples = np.array(vote_samples)

    # 计算每个选手投票的95%置信区间
    lower = np.percentile(vote_samples, 2.5, axis=0)
    upper = np.percentile(vote_samples, 97.5, axis=0)
    mean = np.mean(vote_samples, axis=0)

    return mean, lower, upper
```


问题二： 排名法 vs 百分比法

问题抽象

问题二其实就是一个方法对比分析——分析两种投票合成机制的差异。

> 与问题一的关系：
>
> 问题一用两种方法分别估计了观众投票 ， 得到了两组估计值。
>
> 问题二要做的是： 比较这两组估计 ， 分析两种方法的差异和各自特点。

为什么要对比这两种方法？

这两种方法在数学上有本质区别：

对比维度	排名法	百分比法
信息使用	只用相对排名	使用绝对分数
分数差距	完全忽略	完全保留
优势选手	优势被压缩	优势被放大
冷门选手	更容易逆袭	很难逆袭

举个例子：

假设某周3位选手的评委分是：A=9分 ， B=8分 ， C=5分

用排名法：

- A排第1 ， B排第2 ， C排第3
- 评委分差距（9 vs 5 = 4分）被压缩成排名差距（ 1 vs 3 = 2个

名次）用百分比法：

- A占比 9/(9+8+5) = 41% · B占比 8/22 = 36%
- C占比 5/22 = 23%
- 差距保留：C要翻盘需要比A多18%的投票！

这意味着什么？

如果观众狂热支持C（ 比如C是偶像歌手有粉丝），用排名法C可能逆袭成功 ， 用百分比法C很难翻盘。

所以节目组选择哪种方法 ， 直接影响了"人气选手"能走多远。这就是问题二要分析的核心。

建模思路

数据输入： 问题一估计的两组投票（votes_rank 和 votes_percent）

分析目标：这两种方法会导致多大程度的结果差异？在什么情况下差异最大？

步骤一：对每一季每一周 ， 分别用排名法和百分比法计算综合排

名这是基础对比——看两种方法给出的淘汰人选是否一致。

步骤二： 比较两种方法的淘汰结果是否不同

统计"不一致率" ： 多少周两种方法会淘汰不同的人？

- 如果接近0%：两种方法几乎等价，选哪个无所谓
- 如果超过20%：方法选择很重要，会显著影响比赛结果

步骤三：找出那些"争议选手"——两种方法给出不同结果的案例

这是问题的重点。争议选手的特征是什么？他们往往是：

- 评委分很低但观众很喜欢（粉丝型选手）
- 或者评委分很高但观众不投票（技术型选手）

这样做的好处：

1. 不是空谈理论，而是用数据说话
2. 能找到具体案例支撑论点
3. 为问题四的机制设计提供依据

```
def compare_methods(judge_scores, estimated_votes):
    """比较排名法和百分比法的结果"""
    n = len(judge_scores)

    # 排名法
    judge_ranks = sorted(range(n), key=lambda i: -judge_scores[i])  # 分数高排名小
    vote_ranks = sorted(range(n), key=lambda i: -estimated_votes[i])
    rank_method_scores = [judge_ranks.index(i) + vote_ranks.index(i) for i in
    range(n)] rank_method_elim = max(range(n), key=lambda i:
    rank_method_scores[i])

    # 百分比法
    judge_total = sum(judge_scores)
    judge_pct = [s / judge_total for s in judge_scores]
    percent_method_scores = [0.5 * judge_pct[i] + 0.5 * estimated_votes[i] for i in
    range(n)] percent_method_elim = min(range(n), key=lambda i:
    percent_method_scores[i])

    return {
        'rank_method_elimination': rank_method_elim,
        'percent_method_elimination': percent_method_elim,
        'different': rank_method_elim !=
    percent_method_elim }
```

争议选手分析

根据节目历史[3]，有一些著名的"争议选手"——评委分很低但观众很喜欢。

分析框架：

1. 计算评委排名和观众排名的差距
2. 差距大的就是"争议选手"
3. 分析这些选手在两种方法下的命运差异

```
def find_controversial_contestants(judge_scores_all, votes_all, names_all):
    """找出争议选手"""
    controversies = []

    for week_data in zip(judge_scores_all, votes_all,
        names_all): judges, votes, names = week_data
        n = len(judges)

        judge_ranks = sorted(range(n), key=lambda i: -judges[i])
        vote_ranks = sorted(range(n), key=lambda i: -votes[i])

        for i in range(n):
            j_rank = judge_ranks.index(i)
            v_rank = vote_ranks.index(i)
            divergence = abs(j_rank - v_rank)

            if divergence >= 3:  # 排名差距>=3就算争议
                controversies.append({
                    'name': names[i],
                    'judge_rank': j_rank + 1,
                    'vote_rank': v_rank + 1,
                    'divergence': divergence
                })

    return sorted(controversies, key=lambda x: -x['divergence'])
```

问题三：影响因素分析

问题抽象

问题三其实就是一个回归分析问题——找出影响评委评分和观众投票的关键因素。

输入特征X：年龄、行业、来源地、舞伴
输出标签Y：评委分 / 观众投票 / 最终

名次这个问题的真正价值是什么？

表面上是找影响因素 ， 实际上是要回答一个更深层的问题：

> 评委和观众的判断标准一样吗？

如果年龄对评委分影响大 ， 但对观众投票影响小——说明评委看重"成熟稳重" ， 观众更喜欢"年轻活力 "。

如果舞伴经验对两者影响都很大——说明舞伴选择对比赛结果影响巨大 ， 可能需要在问题四里考虑。

为什么要从"因素分析"入手？

1. 可解释性强：机器学习预测准不准不重要 ， 重要的是能解释"为什么"
2. 支撑问题四：设计新系统需要知道哪些因素最重要
3. 验证问题一：如果影响因素对评委分和估计投票的作用方向相反 ， 说明估计是合理的建模思路

推荐方案：随机森林 + 特征重要性分析

为什么用随机森林而不是线性回归？

方法	优点	缺点
线性回归	系数直观	要求线性关系、特征独立
随机森林	能捕捉非线性、 自动处理交互	黑箱 ， 系数不好解释
XGBoost	精度高	更难解释

选择随机森林的理由：

1. 能处理分类特征（行业、舞伴）：不需要手动one-hot编码
2. 自动给出特征重要性排序：论文里直接用 ， 说服力强
3. 不需要什么假设 ， 简单粗暴：对数据分布没有要求
4. 能发现交互效应： 比如"年轻运动员"和"年轻演员"可能表现不同

```
import pandas as pd
from sklearn.ensemble import
RandomForestRegressor from
sklearn.preprocessing import LabelEncoder
import matplotlib.pyplot as plt


# 读取数据
df = pd.read_csv('data_cleaned.csv')


# 特征编码
le_industry = LabelEncoder()
le_partner = LabelEncoder()
df['industry_encoded'] = le_industry.fit_transform(df['celebrity_industry'])
df['partner_encoded'] = le_partner.fit_transform(df['ballroom_partner'])


# 特征矩阵
X = df[['celebrity_age_during_season', 'industry_encoded', 'partner_encoded',
'season']] y = df['overall_avg_score'] # 目标：评委平均分


# 训练随机森林
rf = RandomForestRegressor(n_estimators=100, random_state=42)
rf.fit(X, y)


# 特征重要性
importances = rf.feature_importances_
feature_names = ['Age', 'Industry', 'Partner', 'Season']


plt.figure(figsize=(10, 6))
plt.barh(feature_names, importances)
plt.xlabel('Importance Score')
plt.title('Feature Importance for Judge Scores')
plt.tight_layout()
plt.savefig('feature_importance.png', dpi=150)
plt.show()
```

生存分析（进阶）

把"被淘汰"当成"事件发生"，用 Cox 比例风险模型分析：

```
from lifelines import CoxPHFitter

# 准备生存分析数据
df['duration'] = df['active_weeks']  # 存活时间=参赛周数
df['event'] = (df['placement'] > 1).astype(int)  # 是否被淘汰（冠军除外）

cph = CoxPHFitter()
cph.fit(df[['duration', 'event', 'celebrity_age_during_season', 'industry_encoded',
       'partner_enc duration_col='duration', event_col='event')

cph.print_summary()
# 系数为正 -> 风险因子，更容易被淘汰
# 系数为负 -> 保护因子，更容易晋级
```

一致性分析

最后比较：这些因素对评委分和观众投票的影响是否一致？

```
# 分别训练两个模型
rf_judge = RandomForestRegressor().fit(X, df['overall_avg_score'])
rf_vote = RandomForestRegressor().fit(X, df['estimated_vote'])  # 问题一估计的投票

# 比较特征重要性
import_judge =
rf_judge.feature_importances_
import_vote =
rf_vote.feature_importances_

# 计算相关性
from scipy.stats import pearsonr
consistency, p_value = pearsonr(import_judge, import_vote)
print(f"评委 vs 观众的因素一致性：{consistency:.3f} (p={p_value:.4f})")
```

问题四：新投票系统设计

问题抽象

问题四其实是一个机制设计问题——设计一套规则，平衡不同利益相关者的诉求。

目标：设计一个既公平（专业性得到体现）又有趣（观众有参与感）的投票系统。

为什么这是一个难问题？

公平性和娱乐性往往是矛盾的：

目标	极端情况	问题
完全公平	只看评委分	观众没有存在感，收视率下降
完全娱乐	只看投票	技术最差的可能夺冠，节目专业性受质疑

历史上DWTS就出现过"争议冠军"——评委分很低但粉丝投票把他抬到冠军。节目口碑受损。

所以我需要设计一个权衡机制，既尊重专业判断，又保持观众热情。

设计原则是什么？

1. 专业兜底：技术太差的选手不应该夺冠，再受欢迎也不行
2. 观众影响力：观众投票要有实质作用，不能沦为摆设
3. 悬念保持：比赛不能一眼看穿结果，要有逆转可能
4. 规则透明：观众能理解规则，知道自己投票有用

建模思路

三种方案，从简单到复杂：

方案一：动态权重机制

核心思想：随比赛进程调整权重

初期（第1-3周）：观众权重大（70%），增加参与感，让更多人关注
后期（第8-10周）：评委权重大（70%），保证冠军实至名归

为什么这样设计？

初期淘汰的都是"弱者"，让观众决定谁走其实影响不大

后期都是强者，需要专业眼光区分高下
渐进变化，观众不会感觉"突然被剥夺权力"

优点：简单、透明、容易解释
缺点：观众可能质疑"后期我的投票没用了"

```
def dynamic_weight_system(week, total_weeks):
    """动态权重：评委权重随周数增加"""
    alpha_start = 0.3  # 第一周评委权重30%
    alpha_end = 0.7    # 最后一周评委权重70%
    alpha = alpha_start + (alpha_end - alpha_start) * (week - 1) / (total_weeks -
    1) return alpha


# 模拟历史数据验证
def simulate_new_system(historical_data):
    results = []
    for season in historical_data:
        new_placements = []
        for week_idx, week_data in enumerate(season):
            alpha = dynamic_weight_system(week_idx + 1, len(season))
            # 用新权重计算淘汰结果
          new_scores = alpha * week_data[ Ijudge_pct I] + (1 - alpha) * week_data[ Ivotes I]
            # ... 计算新的淘汰者
        results.append(new_placements)
    return results
```

方案二： 门槛机制

评委分太低的选手，观众投票打折。

```
def threshold_system(judge_scores, votes, threshold_percentile=25):
    """门槛机制：低于门槛的选手投票打折"""
    threshold = np.percentile(judge_scores, threshold_percentile)

    adjusted_votes = []
    for i, (j, v) in enumerate(zip(judge_scores, votes)):
        if j < threshold:
            adjusted_votes.append(v * 0.5)  # 投票权重减半
        else:
            adjusted_votes.append(v)
```

```
    # 重新归一化
    total = sum(adjusted_votes)
    return [v / total for v in adjusted_votes]
```

方案三：多目标优化（高级）

设计一个同时优化"公平性"和"娱乐性"的系统：

```
from pymoo.core.problem import
ElementwiseProblem from
pymoo.algorithms.moo.nsga2 import NSGA2
from pymoo.optimize import minimize

class VotingSystemProblem(ElementwiseProblem):
    def __init__(self, historical_data):
        # x[0]: 评委权重 (0-1)
        # x[1]: 门槛百分位 (0-100)
        super().__init__(n_var=2, n_obj=2, xl=[0, 0], xu=[1,
        100]) self.data = historical_data

    def _evaluate(self, x, out, *args, **kwargs):
        alpha = x[0]
        threshold_pct = x[1]

        # 目标1：公平性（冠军评委分排名）
        # 分数越高（冠军评委分越高）越好 -> 取负数变成最小化
        fairness = -self.simulate_champion_judge_rank(alpha, threshold_pct)

        # 目标2：娱乐性（争议次数）
        # 争议越少越无聊 -> 取负数
        entertainment = -self.count_controversies(alpha, threshold_pct)

        out["F"] = [fairness, entertainment]
```

```
# 运行NSGA-II
problem = VotingSystemProblem(historical_data)
algorithm = NSGA2(pop_size=100)
res = minimize(problem, algorithm, ('n_gen', 100), verbose=False)


# 画Pareto前沿
import matplotlib.pyplot as plt
plt.scatter(-res.F[:, 0], -res.F[:, 1])  # 取负
数还原plt.xlabel('Fairness (Champion Judge
Rank)')
plt.ylabel('Entertainment (Controversy Count)')
plt.title('Pareto Front: Fairness vs Entertainment')
plt.savefig('pareto_front.png', dpi=150)
```

关键数据发现

根据EDA分析 ， 以下发现对建模很重要：

发现	建模启示
评委分随周数递增（6.5→9.4分）	跨周比较要用相对排名 ， 不能直接比分数
评委间高度一致（相关系数0.85+）	可直接用平均分代表评委意见
运动员和演员容易夺冠	行业是重要控制变量
Derek Hough参赛17季 ， 6次冠军 ， 胜率35%[4]	舞伴效应极其显著
部分赛季仅3位评委（S1-S8）	需分赛季处理评委数据

参考文献

[1] Entertainment Weekly, "How does DWTS voting work? Executive producer explains" https://ew.com/

[2] Rice University, "Dancing with the Stars Voting Analysis" https://www.stat.rice.edu/~dobelman/dwts.html

[3] Collider, "DWTS voting controversy: Why audience votes seem to matter more" https://collider.com/

[4] Cheatsheet, "Derek Hough DWTS Statistics: 6 Wins in 17 Seasons" https://www.cheatsheet.com/

[5] JustJared, "Former DWTS Pro Explains Voting System" https://www.justjared.com/