对生产过程中的决策问题的优化分析

摘 要

本文针对以企业生产零部件以及成品是否检验展开分析和研究，运用多种参数假设检验对抽样检测方案进行优化，采取决策树思想，应用蒙特卡洛模拟处理决策方案，使用并行优化最终给出最小化成本的决策方案。

针对问题一：为零部件是否接收设计检测次数尽可能少的抽样检测方案。由于随机抽样方式结果符合二项分布,得出二项分布概率质量函数及累积分布函数，对二项分布进行假设检验，需找到样本量n及一个临界值k，给出大于k的概率在置信度之内的n,进行 t 检验，随着样本量增大，统计误差显示对于样本量足够大时可以认为近似符合正态分布，进行 **z 检验**得出最佳的样本量n。得到一个理论上的最小检测次数，再进行优化，这时引入**序贯抽样**方案。在保证统计学显著性的前提下，用最少的抽样次数动态来判断零配件的次品率的置信度，从而降低了企业检验成本，提高了检验效率。

针对问题二：为企业生产过程的各个阶段做出是否检验的决策，可建立目标函数，使方案成本最小。对于每一项装备环节，设计**决策树**遍历所有情况，结合次品率要求限制和检测预算限制，采用**动态规划**和**蒙特卡洛模拟**的方法，模拟不同决策路径下的总成本，得出最小成本的决策方案。最后分别对题中的六种情形进行具体比较分析。

针对问题三：是对问题二的拓展延伸。由于规模更大，决策方式及计算量指数上升，需对多工序进行合并简化。故可将零配件组装成半成品的过程，视为问题二的零配件组装成成品的过程，对半成品至成品同理。只需要将问题分为 2 个步骤，将决策树的规模扩展，并采用**并行计算**进行优化。在计算拆解成品和半成品的概率时引入**贝叶斯公式**，通过**蒙特卡洛树搜索**，逐步逼近最优策略，减少需要完整模拟的路径数量。由于数据量庞大，采用树算法短时间无法给出合理决策，可先解决 2 道工序、8 个零配件的情景，再推广到m道工序、n个零配件，最后得出最优决策方案。

针对问题四：本质是对上述所有问题的综合。在实际生活中，由于次品率不确定性及抽样误差的影响，对于问题二和问题三，不同零部件需要通过问题一中 Z 检验的方式确定次品率置信区间，求出次品率的浮动范围，随后将该浮动范围置于正态分布中，重新带入问题二和问题三进行求解。最后将求解后的收益与原收益进行对比分析，并使用**最小二乘法**得出三维的**残差分析**图，为企业给出更优的决策。

总之，本文提出的决策方式在实际企业生产过程中具有现实意义。

关键词：假设检验 决策树 蒙特卡洛模拟 贝叶斯公式 并行计算

一、问题背景

企业决策是经营管理中的关键环节，正确的决策对企业的生存和发展至关重要。为了做出合理的决策，企业应遵循以下五个原则：信息准全原则：决策的科学性依赖于信息的准确性和完整性。这包括收集和分析各类数据，确保数据真实可靠，不含任何虚假成分。可行性原则：决策的正确性需要通过可行性研究来验证。只有那些经过充分论证并在实践中得到验证的决策，才能被认为是科学的。对比选优原则：在做出经营决策时，应提出多种方案，并从中选择最佳方案。这要求对不同方案进行比较分析，以确定最优解。民主决策原则：决策的民主化是科学化的重要前提。在市场经济条件下，决策往往涉及多个方面，需要集合团队成员的智慧和专业部门，尤其是专家的意见，形成一个全面的智囊团。这样可以从多方面听取意见，避免单一视角的局限。独立自主原则：这一原则与民主决策相辅相成。在决策过程中，既要充分考虑团队和专家的意见，也要保持独立思考，确保决策的自主性和创新性[1]。

这些原则共同构成了企业决策的科学化和民主化基础，有助于企业在复杂多变的市场环境中做出正确的决策。企业生产一种电子产品，需要采购多种零配件并装配成成品，半成品。这其中有两个重要因素：采购和装配。对于装配来说次品率成了关键因素，企业可选择报废不合格成品或拆解以回收零配件。问题要求设计最少次数的抽样检测方案，以确定是否接收供应商提供的零配件，要求制定检测和处理决策，分析检测、拆解和市场投放的成本与质量影响。扩展至更复杂的多工序、多零配件生产过程，要求针对具体流程制定决策方案。最后综合则假设次品率通过抽样检测获得，重新分析前述决策。因此企业在零配件的各个环节的产品质量都尤为重视，希望通过科学的数学方法，结合成本、检测费、装配费、销售额的数学模型，对企业的生产过程进行优化。

二、问题的分析

2.1 问题一的分析

为制定一种高效且结果可信的零部件接收检测方案，在减少检测次数的同时，确保检测结论的可靠性。利用二项分布模型来分析和设计检测流程，因为检测结果自然地符合二项分布$X \sim B(n,p)$。基于二项分布的概率质量函数和累积分布函数，能够计算出在给定的检测次数 n 下，观察到特定数量合格或不合格零部件的概率。为设定接收或拒绝零部件的标准。随着检测样本量n的增加，二项分布趋近于正态分布$X \sim N(np, np(1-p))$。可以更精确地界定拒绝域和接收域，即在何种情况下应拒绝或接受零部件，从而制定出更加科学合理的抽样检测方案。该检测方案通过结合二项分布特性和统计检验方法。

得到一个理论上的最小检测次数，设它为最大值，再进行优化，这时引入序贯抽样方案：不事先规定总的抽样个数,而是先抽少量样本，根据其结果，再决定停止抽样或继续抽样、抽多少，这样下去，直至到了上一步理论值停止抽样为止。

运用这种二步优化的方法，在保证统计学显著性的前提下，用最少的抽样次数动态来判断零配件的次品率的置信度，从而降低了企业检验成本，提高了检验效率。

2.2 问题二的分析

为平衡成本最小化与企业信用，需为各阶段制定是否进行检测的决策。这一决策过程综合考虑成本效率与风险评估。若成本不是考量因素，则可以选择全面检测以确保质量；若不考虑企业信用，理论上可以省略所有检测以降低成本。

企业需兼顾两者寻找一个最优的折中方案。需建立一个复杂的目标函数，该函数应涵盖多项成本要素，包括但不限于检测成本、因检测而可能产生的拆解费用、发现次品后的报废损失、以及因质量问题导致的市场调换损失等。为了科学合理地制定决策，可以为生产流程中的每个关键环节设计专属的决策树模型。在构建决策树时，紧密结合各阶段的次品率要求及企业设定的检测预算限制。

采用动态规划技术来优化决策路径，确保在给定资源下做出最优选择。引入蒙特卡洛模拟方法，通过大量随机样本的模拟运行，评估不同决策路径下的总成本，从而更全面地理解不同策略的经济影响。为了验证并优化决策方案，将设置六种不同的生产或市场环境情形，每种情形均代表不同的成本结构、次品率风险及市场反应。通过对这些情形的详细比较与分析，可以识别出在多种条件下均能表现稳健且成本效益最高的决策方案。最终，这将指导企业制定出一套既经济高效又符合信用标准的检测策略。

2.3 问题三的分析

在深入探讨问题三的解决方案时，由于问题规模的显著扩大，直接沿用传统方法将导致决策难度与计算量呈指数级增长，因此，提出了多工序合并简化的策略。将零配件组装成半成品的过程视为一个独立的决策单元，类似于问题二中零配件组装成成品的过程，并进一步将这一逻辑应用于半成品至成品的转换阶段。通过将复杂的生产流程分解为两个核心步骤，并相应地扩展决策树的规模，构建了一个更加全面且精细的决策框架。

引入了并行计算技术，充分利用现代计算资源的并行处理能力，加速决策过程的执行。在计算拆解成品及半成品的概率时，采用了贝叶斯公式，通过融合先验信息与样本数据，实现了对概率分布的精确估计，为决策提供了更加可靠的依据。

为有效减少计算成本并快速逼近最优决策策略，采用了蒙特卡洛树搜索方法。该方法通过模拟不同决策路径下的成本与风险。通过减少需要完整模拟的路径数量，进一步提升了计算效率与决策速度。验证了方法的有效性与可行性。基于成功经验的总结与提炼，将该方法逐步推广至包含m道工序与n个零配件的完整生产环境。通过这一循序渐进的过程，最终得出了适用于大规模生产环境的最优决策方案，为企业的生产决策提供了有力的理论支持与实践指导。

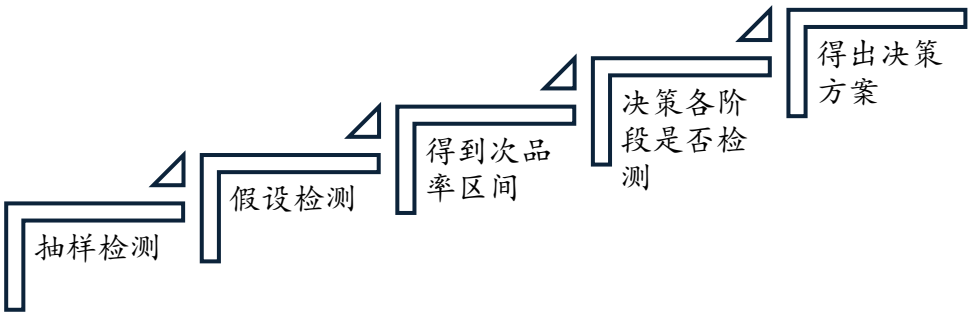
2.4 问题四的分析

在企业的生产决策中，由于次品率的不确定性和抽样过程中可能出现的误差，对于问题二和问题三中提到的不同零部件，需要采用问题一的方法来确定次品率的置信区间。即将次品率限定在一个特定的范围内，并对成本和收益进行详细分析，并增强决策的鲁棒性。若尝试列举所有可能的情况，将会面临极高的复杂度。为了简化这一过程，可以采取将决策步骤聚合的方法，通过合并相似或相关的工序，从而得出一个更为简洁有效的决策方案。将考虑次品率不确定性后的决策结果所带来的预期收益。最后与未考虑此因素时的原收益进行对比分析。

为进一步验证和优化决策，采用最小二乘法构建三维残差分析图。最小二乘

法可以帮助识别决策变量（如生产成本、销售价格、次品率等）与收益之间的非线性关系，并通过残差分析图直观展示实际值与预测值之间的差异，从而为企业提供更为精确和全面的决策支持。

2.5 基本思维导图



三、基本假设

一、假设在抽取零配件时，每个零配件成为次品的可能性是相同的。
二、假设在完好的零配件/半成品组成的成品/半成品拆解时，拆解出的零配件/半成品仍是完好的。
三、生产过程中没有外部因素干扰，如设备故障，市场需求波动。所有操作均不会造成操作效率损失，都能按照规定效率进行。
四、在整个生产周期内，零配件、成品成本价格均固定，不会受外部因素影响，成品损失及拆解费用均固定。
五、假设每次抽样检测的结果都具有代表性。能反应该零配件（样品）的次品率，抽样检测的次品率与总体次品率相等。

四、符号说明

序号	符号	解释
1	p	次品率
2	d	二进制决策变量
3	C	成品价
4	A	装配价
5	B	检测费用
6	S	调换损失费用
7	L	拆解费用
8	M	总量
9	n	样本量

五、模型的建立与求解

5.1 问题一：假设检验模型的求解

5.1.1 模型准备

针对问题一，设计一种检测方案以确定零部件是否接收，目标是在保证结果可信度的前提下，减少检测次数。为实现这一目标，采取假设检验的方式。由于抽取到次品的概率满足二项分布。

首先零假设（H_0）：零配件次品率不超过其标称值$p \le p_0$；和备择假设（H_1）：零配件的次品率高于其标称值 $p > p_0$。根据二项分布概率密度函数的性质，对抽取到次品的概率进行分析，寻找出观测值比标称值更极端的情形。使其在统计学具有实际意义。再通过两种假设检验方式确定检测次数。

得到一个理论上的最小检测次数，设它为最大值，再进行优化，这时引入序贯抽样方案：不事先规定总的抽样个数,而是先抽少量样本，根据其结果，再决定停止抽样或继续抽样、抽多少，这样下去，直至到了上一步理论值，停止抽样。

运用这种二步优化的方法，在保证统计学显著性的前提下，用最少的抽样次数动态来判断零配件的次品率的置信度，从而降低了企业检验成本，提高了检验效率。

5.1.2 二项分布的假设检验

假设供应商这批零部件总量为m，假设抽样次数与检验零部件次数成正比，则从M中抽出n个零部件为样本。从样本m抽出次品的概率$X \sim B(n,p)$.

二项分布概率质量函数：

$$P(x=k) = C_n^k p^k (1-p)^{n-k}$$

累积分布函数：

$$P(x \leqslant k) = \sum_{i=o}^{k} C_n^i p^i (1-p)^{n-i}$$

随着n的增大，X近似服从正态分布 $X \sim N(np, np(1-p))$，累积分布函数可拟合为正态分布概率密度曲线：

$$P(x \leqslant k) = \frac{1}{\sqrt{2\pi np(1-p)}} e^{-\frac{(x-np)^2}{2np(1-p)}}$$

图 1　n 取 20 二项分布和正态分布概率密度曲线

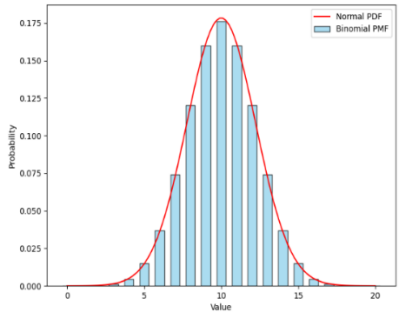
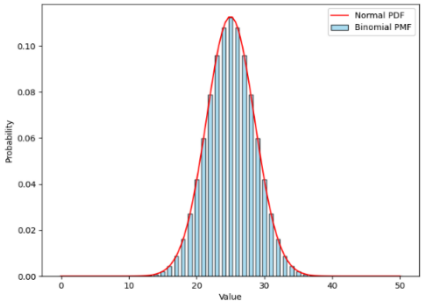


图 2　n 取 50 二项分布和正态分布概率密度曲线



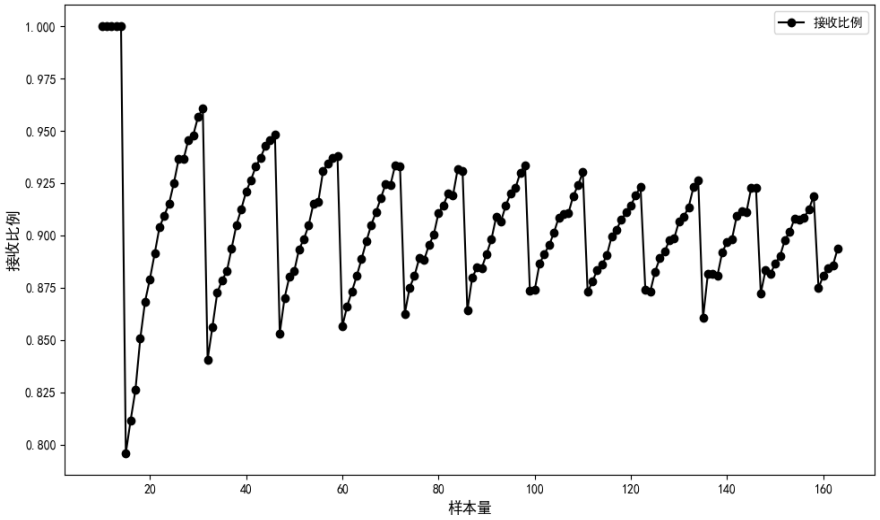
5.1.2.1 Z 检验

对于n较大的情况下，认为$\sigma=\sqrt{np(1-p)}$，简化计算

$$Z=\frac{|p-p_0|}{\sqrt{\frac{p_0(1-p_0)}{n}}}$$

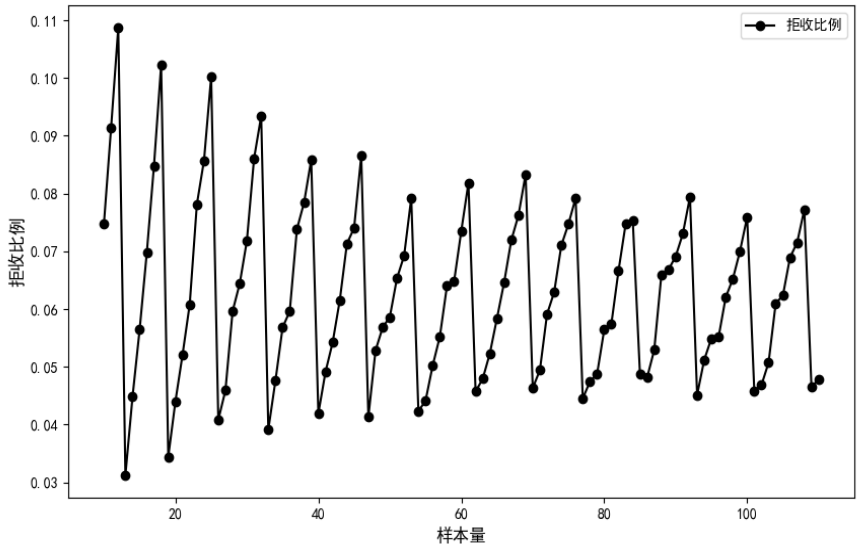
在 95%的信度下认定零配件次品率超过标称值，则拒收这批零配件。

图 3 问题一（1）样本量与接受比例关系



在 90%的信度下认定零配件次品率不超过标称值，则接收这批零配件。

图 4 问题一（2）样本量与接受比例关系



5.1.2.2 T 检验

对于n较小的情况下，由于数量较少，可遍历计算标准差：

$$\sigma=\sqrt{\frac{\sum_{i=1}^{n}(P(x=k)-np)}{n-1}}$$

对于：

$$t = \frac{p - p_0}{\frac{\sigma}{\sqrt{n}}}$$

$$df = n - 1$$

由于抽样过程中存在较大偶然性，为减少随机误差的影响，若在样本量较小的情况下，次品数达到理论值，允许此次实验作废，重新进行实验；若样本在很大的一定范围内，依然没有抽出次品，继续抽取。

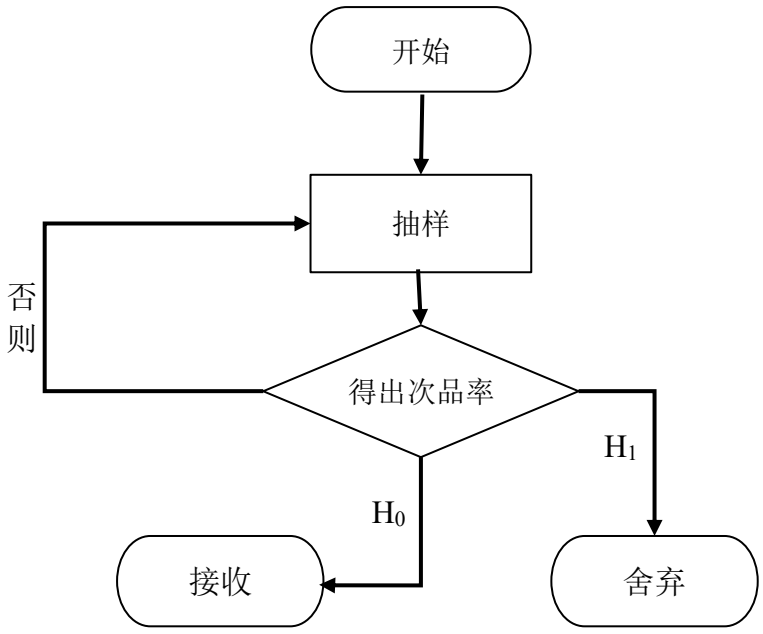
5.1.3 序贯概率比检验

序贯概率比检验的优点在于采用两阶段优化策略，在确保统计显著性的同时，通过最小化抽样次数来动态评估零配件次品率的置信水平，以此降低企业的检验成本并提升检验效率。具体而言：统计学显著性：确保抽样结果在统计学上具有显著性，即抽样得到的次品率估计值与真实次品率之间具有高度的一致性。最小化抽样次数：在满足统计显著性的前提下，通过优化抽样策略，减少必要的抽样次数，从而降低检验成本。动态评估置信度：通过动态调整抽样策略，实时评估零配件次品率的置信度，确保检验结果的准确性和可靠性。提高检验效率：通过上述方法，实现在较低成本下进行高效、准确的检验，从而提高整体的检验效率。

设n为抽样次数，α为显著性水平，p为抽样得到的次品率估计值，$\hat{p}$为真实次品率。目标是在满足$\mathrm{P}(|\hat{p} - p| > \epsilon) \leq \alpha$的条件下，最小化 n，其中ϵ为可接受的误差范围。

通过动态调整抽样策略，实时更新n和p，以确保在最小化检验成本的同时，提高检验效率。

图 5 序贯概率比检验流程



对于零假设（H_0）：零配件次品率不超过其标称值$p \leq p_0$：

$$\frac{P(x = k|p_0)}{P(x = k|p_1)} > \frac{P}{Q}$$

对于备择假设（H_1）：零配件的次品率高于其标称值 $p > p_0$：

$$\frac{P(x=k|p_0)}{P(x=k|p_1)} < Q$$

若：

$$Q \leqslant \frac{P(x=k|p_0)}{P(x=k|p_1)} \leqslant P$$

则继续抽样。

这样动态的调整检测次数，可以在不失置信度的前提下，使检测次数更少。

5.1.4 结果分析

其中s为样本量个数，这里取 10 个样本量（结果生成代码见附录 3）：

在 95%的信度下，满足拒收条件的s:

$$\vec{a}_1 = [14, 15, 20, 21, 27, 33, 34, 35, 40, 41]$$

最优样本量的平均值为:μ_1=28.

在 90%的信度下，满足接收条件的s:

$$\vec{a}_2 = [21, 22, 23, 36, 37, 38, 39, 51, 52, 53]$$

最优样本量的平均值为: μ_2=37.

由于抽样过程中存在较大的偶然性，在样本量较小的情况下，为了减少随机误差的影响，允许在原有样本量的基础上再抽取更多样本进行补充抽样检测。

根据实际生产过程，可采用极差设计检测方案如下表：

抽取次数范围	95%信度下拒收	90%信度下接受
$k \leqslant \frac{s \cdot \mu_i}{\lvert n_{\max} - n_{\min} \rvert}$	若抽取至上界前次品达到理论值，拒收。	若抽取至上界前次品达到理论值，作废，重新实验。
$\frac{s \cdot \mu_i}{\lvert n_{\max} - n_{\min} \rvert} < k \leqslant \mu_i$	若达到下界次品数达到理论值的 50%，连续抽取至上界，直至拒收。	若达到上界次品数小于理论值，接受。
$\mu_i < k \leqslant \frac{\lvert n_{\max} - n_{\min} \rvert \cdot \mu_i}{s}$	若达到上界次品数达到理论值，拒收。	若达到下界次品小于理论值的 50%，连续抽取至上界，直至接受。
$k > \frac{s \cdot \mu_i}{\lvert n_{\max} - n_{\min} \rvert}$	若抽取至下界，次品仍无次品，作废，重新实验。	若抽取至下界，次品未达到理论值，接受。

5.2 问题二：基于蒙特卡洛模拟的单目标规划决策方案

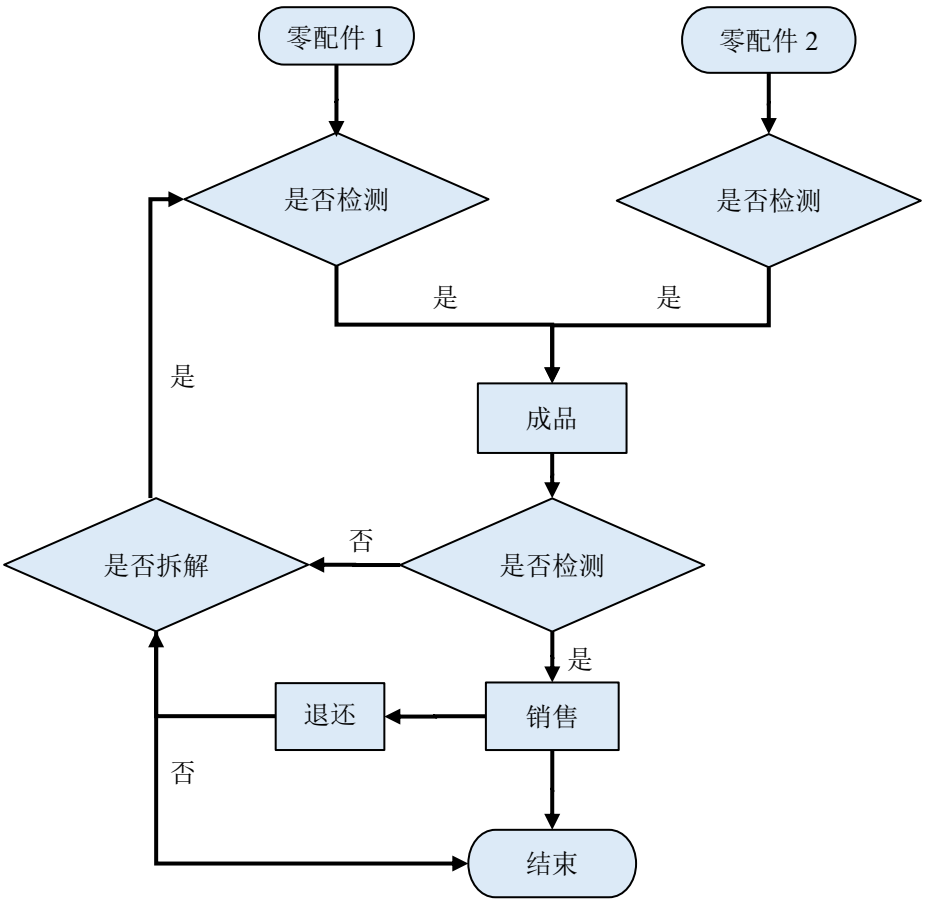
5.2.1 思路的建立

为在企业生产过程中实现成本最小化，同时兼顾企业信用与风险控制，需要构建一个综合决策模型。该模型将基于成本效率分析和风险评估，通过设计针对每个生产环节的决策树，来权衡是否进行检测。成本考虑中，不仅包含直接的检测费用，还涉及潜在的拆解成本、报废损失及市场调换损失等。在决策

过程中，将次品率要求和检测预算作为约束条件，运用动态规划技术优化决策路径，并结合蒙特卡洛模拟方法，模拟不同检测策略下的总成本变动，以期找到在给定条件下成本最低的检测方案。通过对比分析六种不同情景下的模拟结果，将能够识别出最为经济且符合企业信用标准的最佳检测策略，根据此思路作出问题二的流程图。

为了解决这个优化问题，可以构建一个决策树模型，其中每个节点代表一个生产环节的检测决策。利用动态规划和蒙特卡洛模拟来评估不同决策路径下的总成本。动态规划用于在给定的检测预算约束下，递归地计算最小成本路径。蒙特卡洛模拟用于评估在随机次品率情况下的预期成本。

图 6 问题二流程图



1、是否零配件检测：

- 不检测零配件：若零配件次品率较低，则可以选择不进行检测零配件次品率，但要承担成品次品率上升问题，使后续步骤复杂化。
- 检测零配件：若零配件次品率较高，则可对这一零配件进行检测，使成品次品率下降，但要承担零配件检测费用。

2、是否成品检测：

- 不检测成品：若零配件质量较好次品率较低且成品次品率较低，则可选择不检测成品，直接进行销售，以降低成本，但要承担一定的调换风险。
- 检测成品：若零配件次品率较高或成品次品率较高，则应对成品进行检测，来规避调换或拆解成本风险。

3、是否拆解：

- 不进行拆解：当成品成本或者利润低廉时，则可直接进行舍弃。

● 进行拆解：对于成品成本和拆解费用相差悬殊时，可选择进行拆解，这一步骤循环利用，但是浪费了相应的拆解成本及次品的组装成本。

5.2.2 目标规划的构建

d_1, d_2：是否对零配件 1 和零配件 2 进行检测，d_3：是否对装配成品进行检测，D_r：是否对检测出的不合格成品进行拆解。$d_1, d_2, d_3, D_r \in \{0,1\}$

设p_1, p_2, p_3分别为零配件 1、零配件 2、成品的次品率，C为成品单价，B_1，B_2，B_3为检测成本，A_1, A_2, A_3为零配件 1、零配件 2 成本及成品的装配价，S为调换损失费用，L为拆解费用。对于采购N件成品。

零部件费用：零配件采购数量、零配件采购成本、零配件检测费用

$$\begin{cases} N_i = \dfrac{N}{1 - p_i(1 - d_i)} \\ S_i = A_i N_i \\ S_3 = d_i \cdot \dfrac{N}{1 - p_i} \cdot B_1 \end{cases}$$

其中N为零配件的采购数量、S_i零配件采购成本为上述(i=1, 2)、S_3零配件检测费用.

成品装配费用：装配成本、检测成本、调换损失、拆解费用、成品销售额.

$$\begin{cases} S_4 = A_3 N \\ S_5 = d_3 N B_3 \\ S_6 = NS \sum_{i=1}^{3} (1 - p_i)(1 - d_i) \\ S_7 = LD_r N \sum_{i=1}^{3} (1 - p_i)(1 - d_i) \\ W = CN \end{cases}$$

其中S_4为装配成本、S_4 检测成本、S_5 调换损失、S_6拆解费用、W成品销售额.

目标函数为抛去零配件成本的其他处理成本：

$$W - \sum_{i=3}^{7} S_i$$

约束条件：

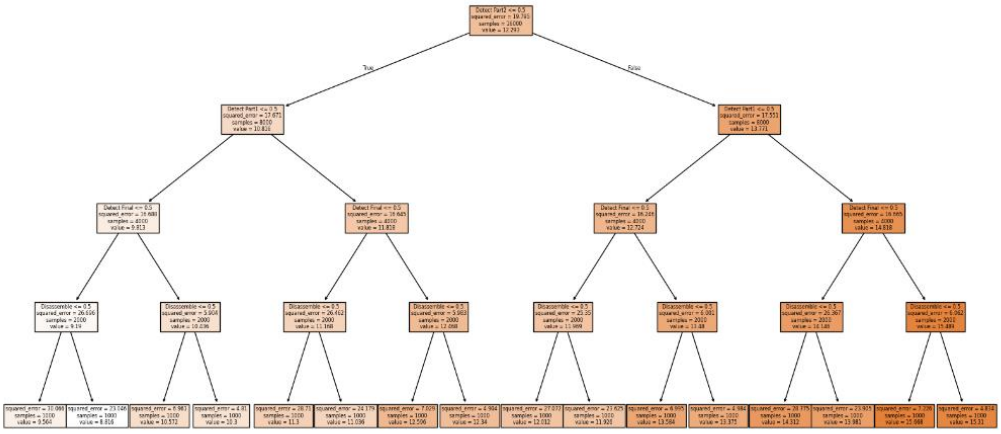
$$p_i \leq p_0$$

（这里d_i为判断零部件是否检测，D_i为判断（半）成品是否检测）

5.2.3 结果呈现

情况一最优成本决策树图结果（完整决策树图见附录 2）：

图 7 运行情况 1 的决策树工程图

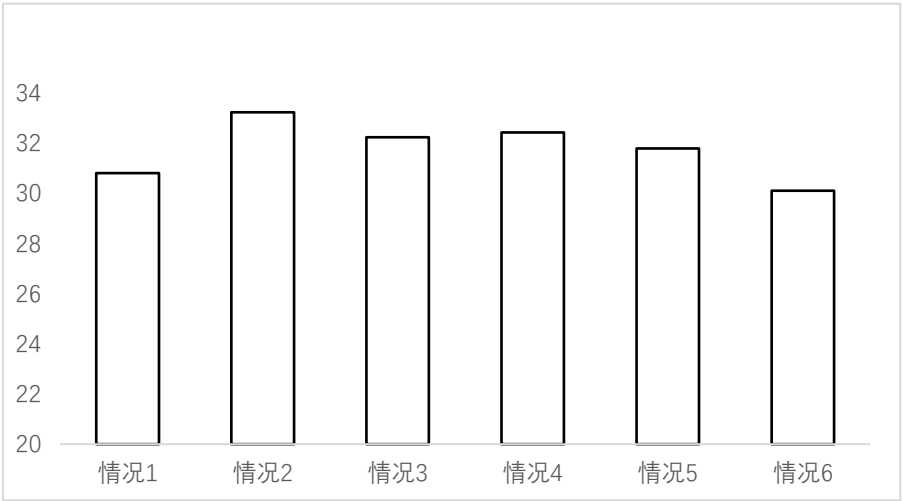


问题二六种情况结果汇总（结果生成代码见附录 3）：

	情况 1	情况 2	情况 3	情况 4	情况 5	情况 6
d_1	0	0	0	1	0	0
d_2	0	0	0	1	1	0
d_3	0	0	1	1	1	0
D_r	1	1	1	1	1	0
W	8.82	11.24	10.25	10.44	9.80	8.11

（W为除去零配件成本的其他处理成本）

图 8 六种情况的单件成品利润



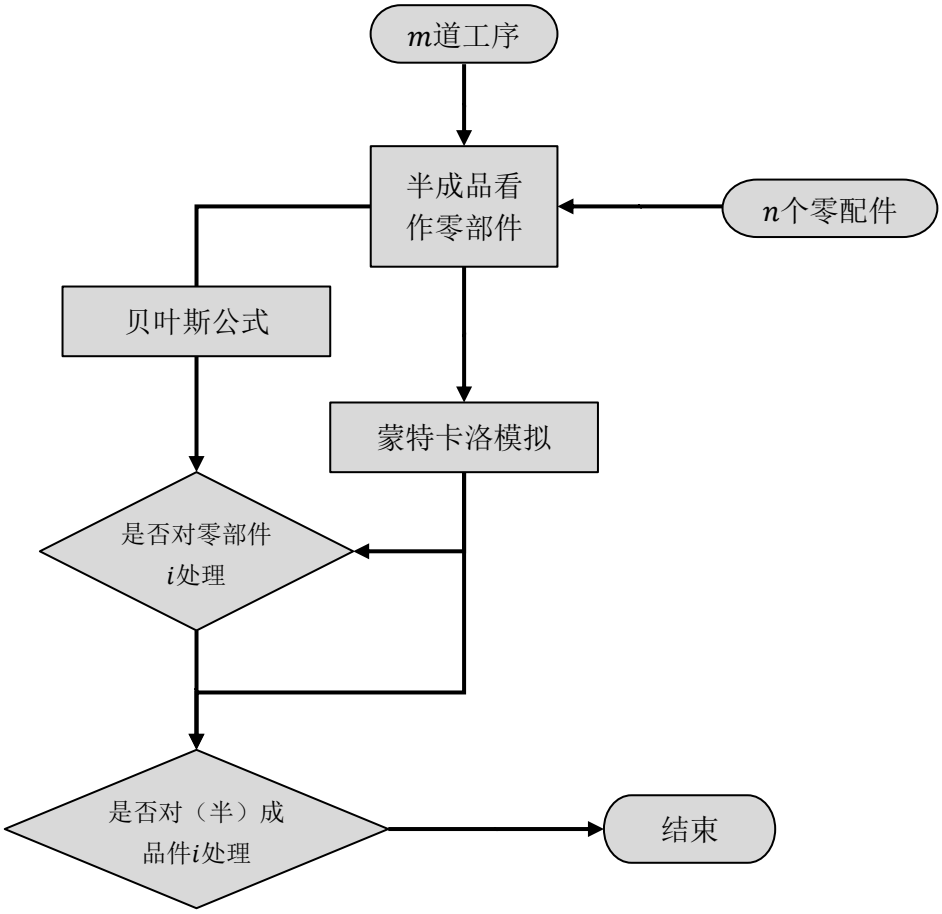
可见，在成品市场售价基本相近情况下，如若调换损失越大，则越倾向于对成品检测；若拆解费用相对成本悬殊，则趋向于拆解；至于对零配件是否检测给出方案，需综合根据零配件及成品次品率来决定。

5.3 问题三：复杂工序的简化搜索模型

5.3.1 简化思路

针对问题三，采取了一种分层次分步骤的方法来处理涉及 m 道工序和 n 个零配件的复杂生产决策问题。通过合并多个工序来简化决策过程，并将其分解为两个主要步骤：将零配件组装成半成品，进一步组装成成品。为了适应更大规模的问题，扩展了决策树的规模，并利用并行计算来提高计算效率。在评估拆解成本和零配件状态概率时，引入了贝叶斯公式来提高准确性。采用蒙特卡洛树搜索算法来逐步逼近最优策略，减少需要完整模拟的路径数量。由于直接处理大规模问题可能不现实，先从较小规模的问题开始，逐步推广到更复杂的场景。通过这些策略，能够有效地处理大规模生产决策问题，并最终得出合理的结论和决策方案。

图 9 问题三流程图



5.3.2 数据处理

在处理问题三时，可将零配件组装成半成品的过程视为问题二的零配件组装成成品的过程，对于半成品至成品同理。只需要将问题分为 2 个步骤，并将决策树的规模扩展，在计算拆解成品和半成品时应用全概率贝叶斯公式，由于情况过于复杂，采用并行计算，先得到 2 个步骤八个零配件的决策方案，随后将问题的步骤扩展至m道工序、n个零配件。

半成品的修正后成本：

$$\begin{cases} N_i = \dfrac{N}{1-p_i(1-d_i)} \\ S_i = A_i N_i \\ P_1 = d_i \cdot \dfrac{N}{1-p_i} \cdot B_1 \\ P_2 = A_3 N \\ P_3 = d_3 N B_3 \\ P_4 = NS\sum_{i=1}^{3}(1-p_i)\,(1-d_i) \\ P_5 = LD_r N\sum_{i=1}^{3}(1-p_i)\,(1-d_i) \end{cases}$$

$$G_N = \sum_{i=1}^{5} P_i - \sum_{i=1}^{k} S_k$$

其中G_N为半成品的修正后成本。N=1, 2, …, $\left[\dfrac{n}{2^m}\right]$.

利润：

$$W = CN - \sum_{i=1}^{N} G_N$$

在拆解（半）成品时引入贝叶斯公式：

$$P(A_i|B) = \frac{P(B|A_i)P(A_i)}{\sum_{j=1}^{J} P(B|A_j)P(A_j)}$$

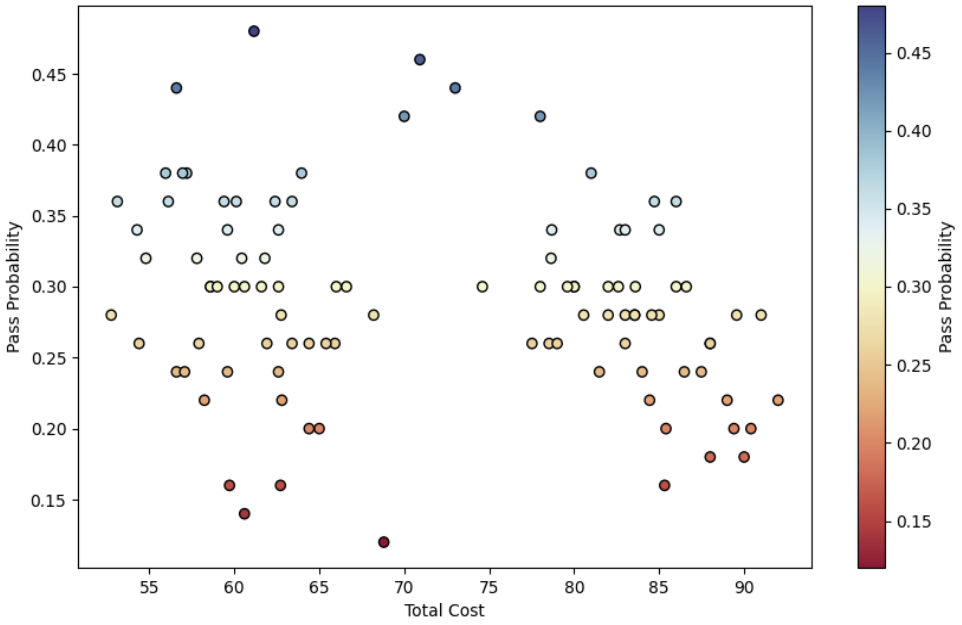
5.3.3 结果分析

最优决策（结果生成代码见附录 3）：

	是否检测		是否检测	是否拆解
零配件 1	0	半成品 1	0	0
零配件 2	1	半成品 2	0	0
零配件 3	1	半成品 3	0	0
零配件 4	0	成品	1	0
零配件 5	1			
零配件 6	1			
零配件 7	1			
零配件 8	1			

对应的最小总成本: 49.23.

图 10 贝叶斯分析图



为解决m道工序、n道零配件的问题，根据问题三所提供的表格，将零配件、中间成品和成品的次品率等简化为如下表格：

所有零配件：

次品率	购买单价	检测成本
10%	8	1.375

所有中间成品：

次品率	装配成本	检测成本	拆解费用
10%	8	4	6

成品：

次品率	装配成本	检测成本	拆解费用	市场售价	调换损失
10%	8	4	6	$25n$	$5n$

如输入m=3，n=4 时，可以得出最小成本为：56.65 元.（结果生成代码见附录 3）

5.4 问题四：不确定性数据的优化模型

5.4.1 不确定性处理

生产中，次品率的不确定性及抽样误差的存在对问题二和问题三提出了新的挑战。这一问题的核心即为数据有了不可靠性，需要企业另外对数据的质量进行分析，确保数据可以真实反应质量。为了应对这些不确定性，可以将问题一的方式应用于不同零部件，以确定次品率的置信区间。

鉴于实际情况的复杂性，可以通过聚合工序的方法简化决策过程。这涉及到将多个连续的生产步骤合并为一个简化的决策节点，从而降低模型的复杂度，使得决策过程更加高效和实用。

根据：

$$Z = \frac{|p - p_0|}{\sqrt{\frac{p_0(1-p_0)}{n}}}$$

得出：

$$p \in \left(p_0 - Z \cdot \sqrt{\frac{p_0(1-p_0)}{n}},\ p_0 + Z \cdot \sqrt{\frac{p_0(1-p_0)}{n}} \right)$$

5.4.2 重新决策规划上述问题

将次品率p置入一个置信区间。

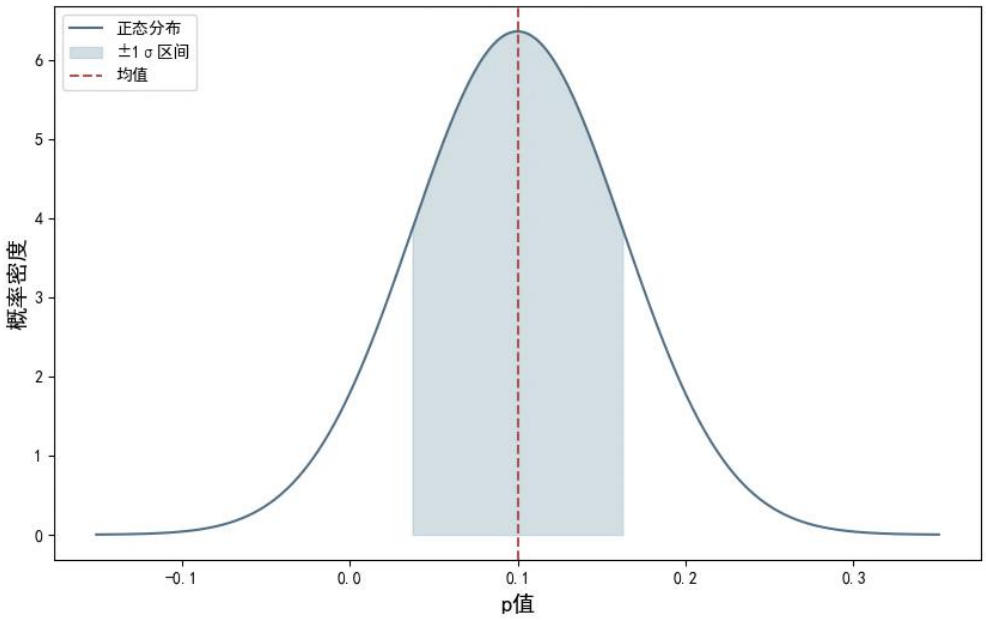
假设次品率$p_0 = 0.1$，并且检测数据的可信度为$r = 99\%$，查表可知 Z 值为 2.576（总表见附录 1）；分类讨论拟定在不同的样本量下置信区间发生的变化。由问题一的检验结果设置：$n \in (10, 160)$

可得：

$$p \in (0.1 - 0.06269088936839551,\ 0.1 + 0.06269088936839551)$$

可视化表述如图 11：

图 11 p 值正态分布图



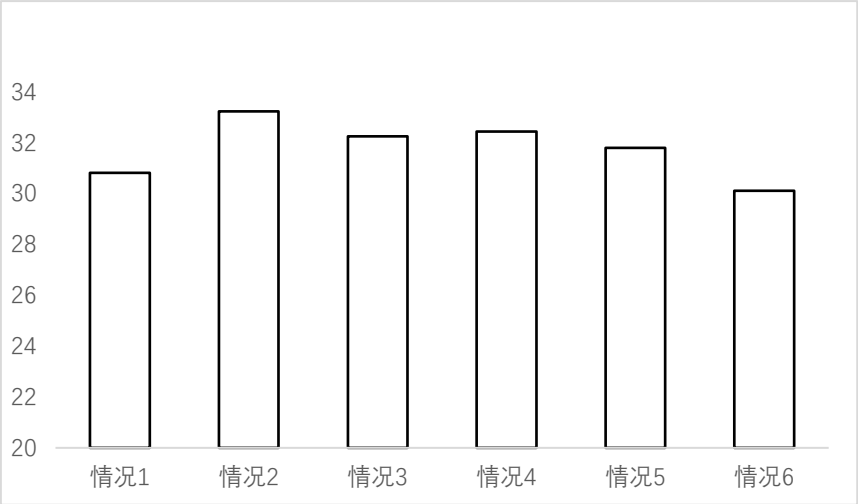
在问题 2 和问题 3 的代码中实现如下：

```
# 引入误差的次品率，通过正态分布模拟
def apply_defect_rate_with_error(defect_rate, error_range=0.06269088936839551):  10 usages
    # 使用正态分布进行浮动模拟
    return np.random.normal(defect_rate, error_range / 2)
```

得出修正过的问题二结果（此后结果生成代码均见附录 3）

	情况 1	情况 2	情况 3	情况 4	情况 5	情况 6
d_1	0	0	0	1	0	0
d_2	0	0	0	1	1	0
d_3	0	0	1	1	1	0
D_r	1	1	1	1	1	0
W	8.78	11.26	10.38	10.54	9.76	8.40

图 12 修正过的六种情况的单件成品利润



得出修正过的问题三结果：

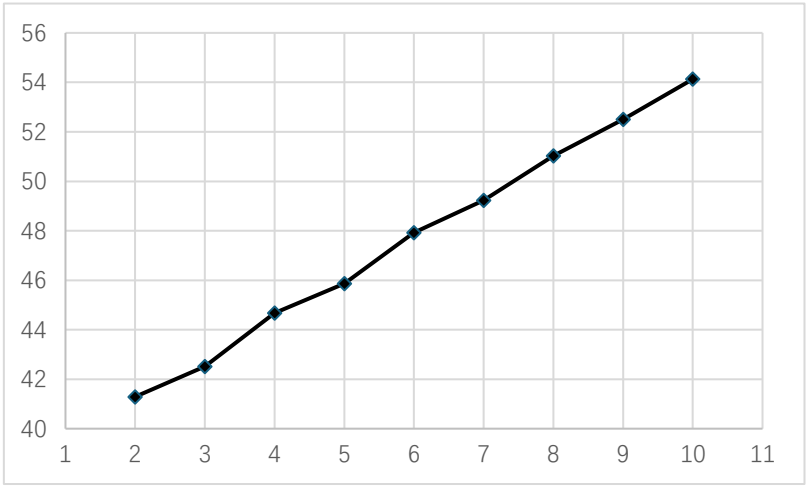
	是否检测		是否检测	是否拆解
零配件 1	0	半成品 1	0	0
零配件 2	1	半成品 2	0	0
零配件 3	1	半成品 3	1	1
零配件 4	0	成品	1	0
零配件 5	1			
零配件 6	1			
零配件 7	1			
零配件 8	1			

对应的成品平均最小总成本: 51.34.

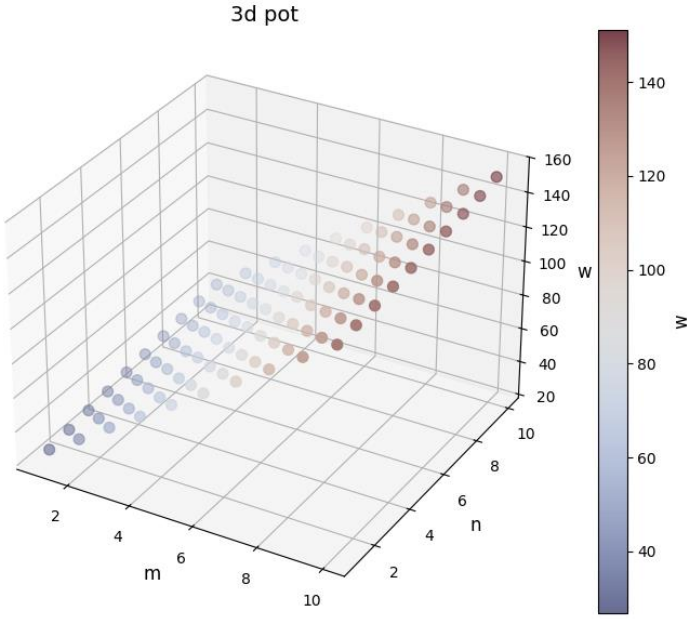
输入m=3，n=4 时，最小利润为：57.14.

输入m=4，n=8 时，最小利润为：75.33.

图 13 m 取 2 时，利润随 n 的变化趋势



循环多次，得出工序道数m、零配件个数n与利润的三维散点图：



5.4.3 误差分析

对于图 10，采用线性回归方程拟合。

线性回归方程对于数据(x_1, y_1)，(x_2, y_2)，…，(x_n, y_n)有$\hat{y} = \hat{b}x + \hat{a}$

$$\begin{cases} \hat{b} = \dfrac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{n}(x_i - \overline{x})^2} \\ \hat{a} = \overline{y} - \hat{b}\overline{x} \end{cases}$$

残差：

$$\hat{\varepsilon} = y - \hat{y}$$

计算得出：

$$\hat{w} = 38.64 + 1.46n$$

依据此思路，用最小二乘法，线性回归分析，给出用平面拟合。

$$\hat{w} = \hat{\beta}_0 + \hat{\beta}_1 m + \hat{\beta}_2 n$$

用矩阵形式表示最小二乘法的解法：

$$\mathbf{W} = \mathbf{X} \cdot \beta$$

其中矩阵 X 是包含 m,n 和常数项 1 的矩阵，$\beta = [\beta_0, \beta_1, \beta_2]^T$为回归系数向量，$W$ 为因变量ω的向量.

$$\mathbf{X}=\begin{bmatrix} m_1 & n_1 & 1 \\ m_2 & n_2 & 1 \\ \vdots & \vdots & \vdots \\ m_N & n_N & 1 \end{bmatrix}$$

$$\mathbf{W}=[w_1,\ w_2,\cdots,w_N]^T$$

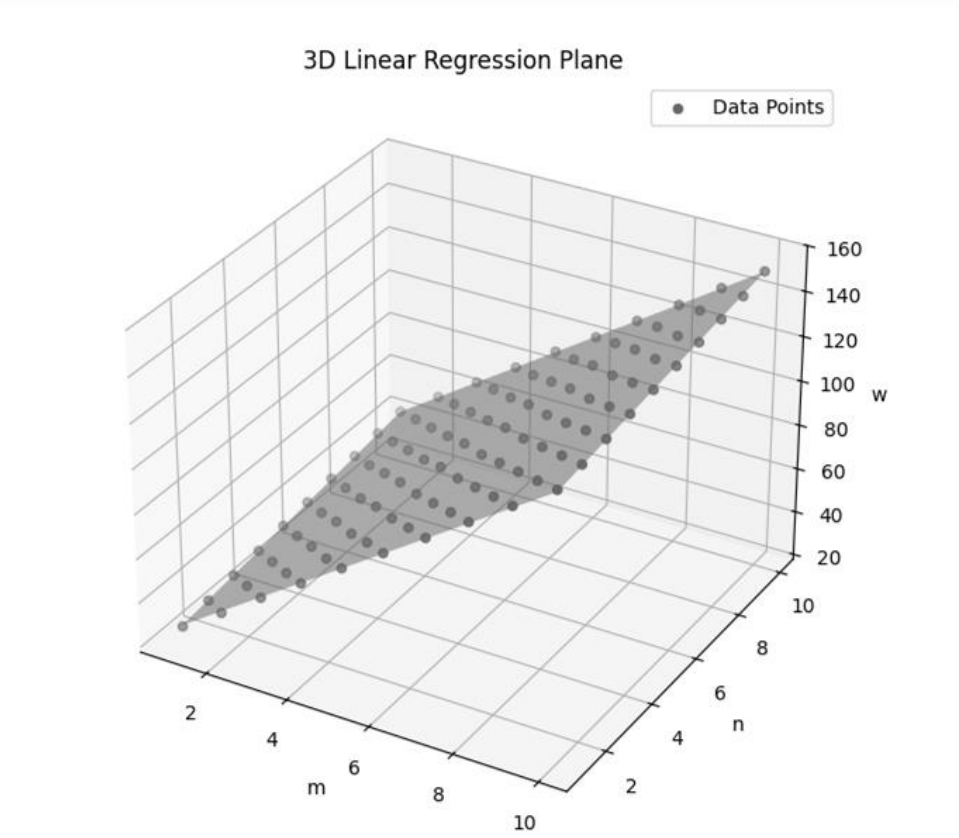
对 β：

$$\beta=(\mathbf{X^T X})^{-1}\mathbf{X}^T\mathbf{W}$$

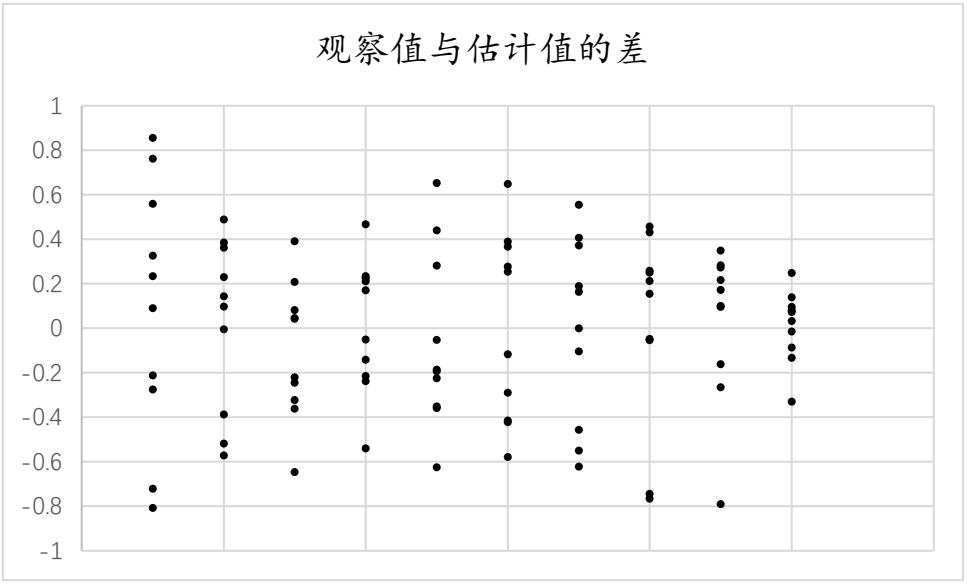
进而求出 β，得到 β_0、β_1、β_2。
得到；

$$\hat{w}=12.1271m+1.5870n+13.8672$$

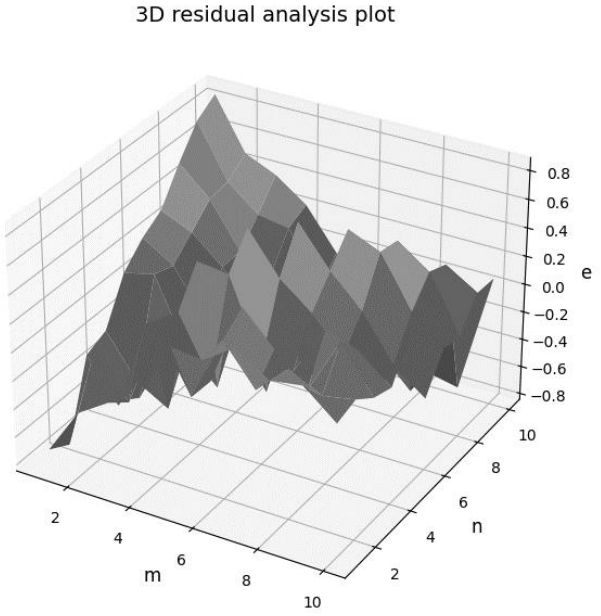
绘制出平面图：



残差分析如下表：



三维可视化如下：



（误差分析的所有图及生成代码见附件）

六、模型的评价与推广

6.1 模型的优点

- 灵活性强。可根据不同的零配件，不同工序，不同的生产环境，分别给出合理的决策，具体问题具体分析。
- 简化思路。在第三问中，将半成品看作零配件，半成品装配费及拆解费减去拆解后零配件成本视为半成品成本，运用转化的思想简化模型。
- 可靠性。运用决策树的思想，分析讨论了不同环境下的企业决策问题。
- 不依赖于分布假设，许多决策要求数据符合特定的分布假设，该模型可以使

用任何分布进行采样，更加灵活

- 适应性强。可以根据需要调整算法，例如通过改变部分数据，来提高计算结果的精确度。
- 可视化。可以方便生成样本数据并进行可视化，从而帮助理解复杂的随机过程或系统。

6.2 模型的缺点

- 缺少实际数据支持，没有真实案例。在实际生产生活中次品率和售价会随着市场变化而改变，未考虑数据波动，本题假设过于理想化
- 过于依赖次品率这一数据源。几乎所有计算全部依赖这个书数据，如果次品率这一数据计算有误，会影响整个决策方案。

6.3 模型的改进

引入实时数据更新，或调查企业的真实数据，来解决市场数据波动问题。

6.4 模型的推广

该决策模型可以推广到其他领域，比如农业食品检测与是否回收问题、金融上股票分析。问题一中的假设检验可以推广到二元变量，如本题可继续计算零配件与成品的次品率关系。

七、参考文献

[1]朱爱华.卡尔曼滤波和序贯概率比检验在管道泄漏监测中的应用[D].天津大学,2006.李牧原,罗德海.冬季北极增暖与中纬度环流和极寒天气之间的联系:经向位涡梯度的 关键作用[J].中国科学:地球科学, 1335-1345，2019,

[2]刘昆朋,曾庆华.序贯概率比检验用于残差检测的一种改进方法[J].电光与控制,2009,16(08):36-39.刘政阳,李挺宇.全球气候变暖趋势急剧加速[J].生态经济, 1-4.2019,

[3]刘俊,罗永峰.钢结构现场检测计数抽样方法研究[J].建筑钢结构进展,2019,21(05):33-39.DOI:10.13969/j.cnki.cn31-1893.2019.05.005.程立勋.浅议企业的经营决策[J].黑龙江科技信息,2002,(07):43.

附录 1

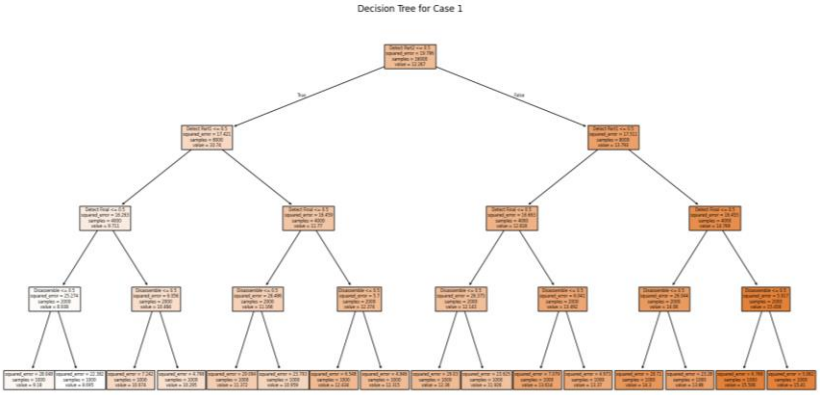
Z 检验标准表

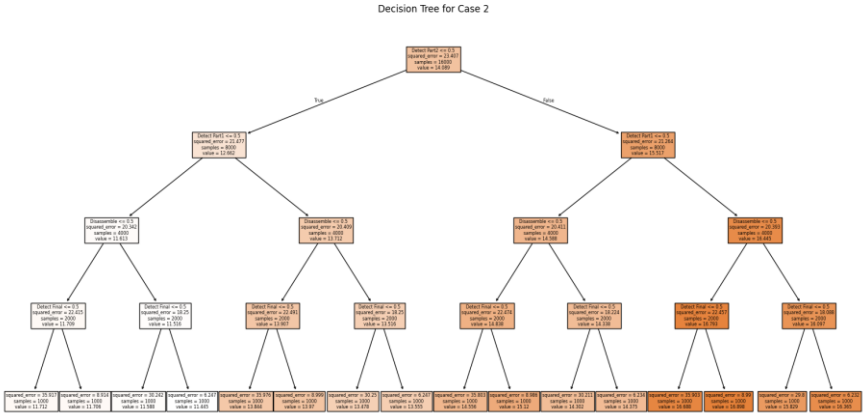
z	.00	.01	.02	.03	.04	.05	.06	.07	.08	.09
–3.4	.0003	.0003	.0003	.0003	.0003	.0003	.0003	.0003	.0003	.0002
–3.3	.0005	.0005	.0005	.0004	.0004	.0004	.0004	.0004	.0004	.0003
–3.2	.0007	.0007	.0006	.0006	.0006	.0006	.0006	.0005	.0005	.0005
–3.1	.0010	.0009	.0009	.0009	.0008	.0008	.0008	.0008	.0007	.0007
–3.0	.0013	.0013	.0013	.0012	.0012	.0011	.0011	.0011	.0010	.0010
–2.9	.0019	.0018	.0018	.0017	.0016	.0016	.0015	.0015	.0014	.0014
–2.8	.0026	.0025	.0024	.0023	.0023	.0022	.0021	.0021	.0020	.0019
–2.7	.0035	.0034	.0033	.0032	.0031	.0030	.0029	.0028	.0027	.0026
–2.6	.0047	.0045	.0044	.0043	.0041	.0040	.0039	.0038	.0037	.0036
–2.5	.0062	.0060	.0059	.0057	.0055	.0054	.0052	.0051	.0049	.0048
–2.4	.0082	.0080	.0078	.0075	.0073	.0071	.0069	.0068	.0066	.0064
–2.3	.0107	.0104	.0102	.0099	.0096	.0094	.0091	.0089	.0087	.0084
–2.2	.0139	.0136	.0132	.0129	.0125	.0122	.0119	.0116	.0113	.0110
–2.1	.0179	.0174	.0170	.0166	.0162	.0158	.0154	.0150	.0146	.0143
–2.0	.0228	.0222	.0217	.0212	.0207	.0202	.0197	.0192	.0188	.0183
–1.9	.0287	.0281	.0274	.0268	.0262	.0256	.0250	.0244	.0239	.0233
–1.8	.0359	.0351	.0344	.0336	.0329	.0322	.0314	.0307	.0301	.0294
–1.7	.0446	.0436	.0427	.0418	.0409	.0401	.0392	.0384	.0375	.0367
–1.6	.0548	.0537	.0526	.0516	.0505	.0495	.0485	.0475	.0465	.0455
–1.5	.0668	.0655	.0643	.0630	.0618	.0606	.0594	.0582	.0571	.0559
–1.4	.0808	.0793	.0778	.0764	.0749	.0735	.0721	.0708	.0694	.0681
–1.3	.0968	.0951	.0934	.0918	.0901	.0885	.0869	.0853	.0838	.0823
–1.2	.1151	.1131	.1112	.1093	.1075	.1056	.1038	.1020	.1003	.0985
–1.1	.1357	.1335	.1314	.1292	.1271	.1251	.1230	.1210	.1190	.1170
–1.0	.1587	.1562	.1539	.1515	.1492	.1469	.1446	.1423	.1401	.1379
–0.9	.1841	.1814	.1788	.1762	.1736	.1711	.1685	.1660	.1635	.1611
–0.8	.2119	.2090	.2061	.2033	.2005	.1977	.1949	.1922	.1894	.1867
–0.7	.2420	.2389	.2358	.2327	.2296	.2266	.2236	.2206	.2177	.2148
–0.6	.2743	.2709	.2676	.2643	.2611	.2578	.2546	.2514	.2483	.2451
–0.5	.3085	.3050	.3015	.2981	.2946	.2912	.2877	.2843	.2810	.2776
–0.4	.3446	.3409	.3372	.3336	.3300	.3264	.3228	.3192	.3156	.3121
–0.3	.3821	.3783	.3745	.3707	.3669	.3632	.3594	.3557	.3520	.3483
–0.2	.4207	.4168	.4129	.4090	.4052	.4013	.3974	.3936	.3897	.3859
–0.1	.4602	.4562	.4522	.4483	.4443	.4404	.4364	.4325	.4286	.4247
–0.0	.5000	.4960	.4920	.4880	.4840	.4801	.4761	.4721	.4681	.4641

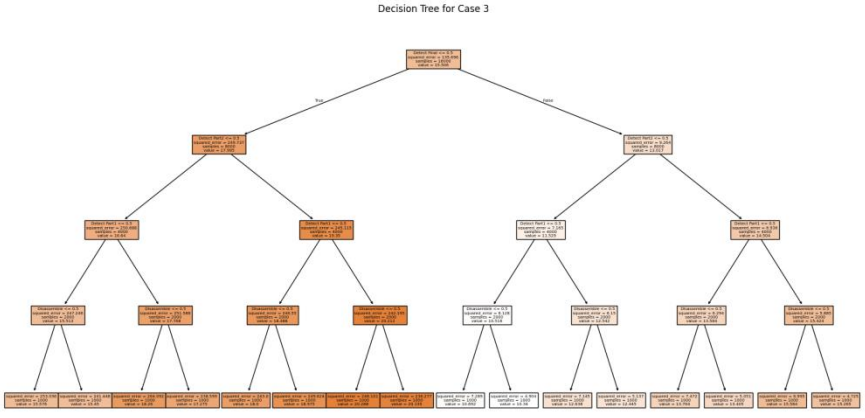
z	.00	.01	.02	.03	.04	.05	.06	.07	.08	.09
0.0	.5000	.5040	.5080	.5120	.5160	.5199	.5239	.5279	.5319	.5359
0.1	.5398	.5438	.5478	.5517	.5557	.5596	.5636	.5675	.5714	.5753
0.2	.5793	.5832	.5871	.5910	.5948	.5987	.6026	.6064	.6103	.6141
0.3	.6179	.6217	.6255	.6293	.6331	.6368	.6406	.6443	.6480	.6517
0.4	.6554	.6591	.6628	.6664	.6700	.6736	.6772	.6808	.6844	.6879
0.5	.6915	.6950	.6985	.7019	.7054	.7088	.7123	.7157	.7190	.7224
0.6	.7257	.7291	.7324	.7357	.7389	.7422	.7454	.7486	.7517	.7549
0.7	.7580	.7611	.7642	.7673	.7704	.7734	.7764	.7794	.7823	.7852
0.8	.7881	.7910	.7939	.7967	.7995	.8023	.8051	.8078	.8106	.8133
0.9	.8159	.8186	.8212	.8238	.8264	.8289	.8315	.8340	.8365	.8389
1.0	.8413	.8438	.8461	.8485	.8508	.8531	.8554	.8577	.8599	.8621
1.1	.8643	.8665	.8686	.8708	.8729	.8749	.8770	.8790	.8810	.8830
1.2	.8849	.8869	.8888	.8907	.8925	.8944	.8962	.8980	.8997	.9015
1.3	.9032	.9049	.9066	.9082	.9099	.9115	.9131	.9147	.9162	.9177
1.4	.9192	.9207	.9222	.9236	.9251	.9265	.9279	.9292	.9306	.9319
1.5	.9332	.9345	.9357	.9370	.9382	.9394	.9406	.9418	.9429	.9441
1.6	.9452	.9463	.9474	.9484	.9495	.9505	.9515	.9525	.9535	.9545
1.7	.9554	.9564	.9573	.9582	.9591	.9599	.9608	.9616	.9625	.9633
1.8	.9641	.9649	.9656	.9664	.9671	.9678	.9686	.9693	.9699	.9706
1.9	.9713	.9719	.9726	.9732	.9738	.9744	.9750	.9756	.9761	.9767
2.0	.9772	.9778	.9783	.9788	.9793	.9798	.9803	.9808	.9812	.9817
2.1	.9821	.9826	.9830	.9834	.9838	.9842	.9846	.9850	.9854	.9857
2.2	.9861	.9864	.9868	.9871	.9875	.9878	.9881	.9884	.9887	.9890
2.3	.9893	.9896	.9898	.9901	.9904	.9906	.9909	.9911	.9913	.9916
2.4	.9918	.9920	.9922	.9925	.9927	.9929	.9931	.9932	.9934	.9936
2.5	.9938	.9940	.9941	.9943	.9945	.9946	.9948	.9949	.9951	.9952
2.6	.9953	.9955	.9956	.9957	.9959	.9960	.9961	.9962	.9963	.9964
2.7	.9965	.9966	.9967	.9968	.9969	.9970	.9971	.9972	.9973	.9974
2.8	.9974	.9975	.9976	.9977	.9977	.9978	.9979	.9979	.9980	.9981
2.9	.9981	.9982	.9982	.9983	.9984	.9984	.9985	.9985	.9986	.9986
3.0	.9987	.9987	.9987	.9988	.9988	.9989	.9989	.9989	.9990	.9990
3.1	.9990	.9991	.9991	.9991	.9992	.9992	.9992	.9992	.9993	.9993
3.2	.9993	.9993	.9994	.9994	.9994	.9994	.9994	.9995	.9995	.9995
3.3	.9995	.9995	.9995	.9996	.9996	.9996	.9996	.9996	.9996	.9997
3.4	.9997	.9997	.9997	.9997	.9997	.9997	.9997	.9997	.9997	.9998

附录 2

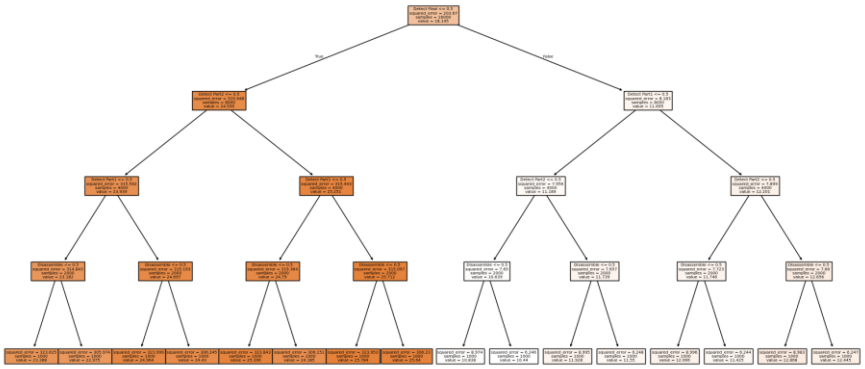
问题二 6 种情况的决策树图



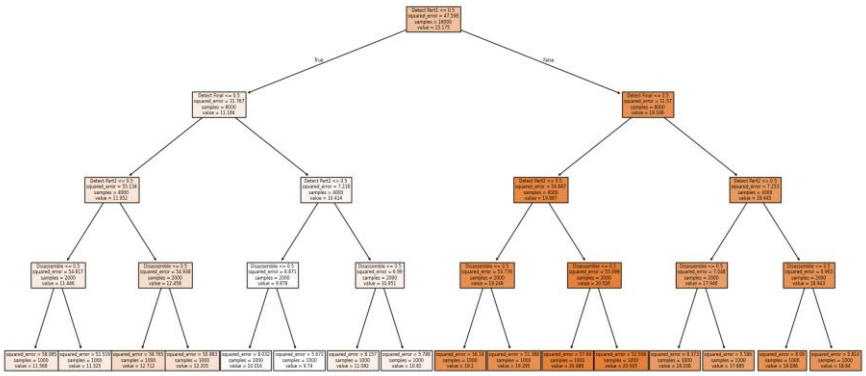




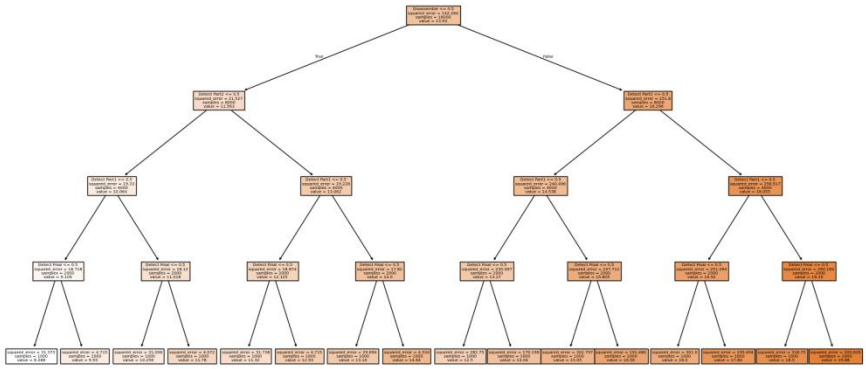
Decision Tree for Case 4



Decision Tree for Case 5



Decision Tree for Case 6



附录 3

问题一的求解：

```
    import numpy as np
    import scipy.stats as stats

    if __name__=="__main__":
        # 次品率
        p_nominal = 0.10

        confidence_reject = 0.95  # 拒收信度
        confidence_accept = 0.90  # 接收信度

        # Z 值对应的临界值
        z_reject = stats.norm.ppf(confidence_reject)  # 95 拒收 Z
        z_accept = stats.norm.ppf(1 - confidence_accept)  # 90%
拒收 Z

        tolerance = 0.01  # ±1%的浮动


        # 随机抽样
        def perform_test(n_samples, p_nominal, z_reject=None,
z_accept=None, num_trials=10000, fluctuation=0.01):
            reject_count = 0
            accept_count = 0

            for _ in range(num_trials):

                # 浮动
                p_fluctuated = p_nominal + np.random.uniform(-
fluctuation, fluctuation)
                p_fluctuated = max(0, min(1, p_fluctuated))  #
确保次品率在 [0, 1] 范围内

                # 次品浮动率
                defective_samples =
np.random.binomial(n_samples, p_fluctuated)

                # 样本次品率
                p_sample = defective_samples / n_samples

                # 拒收
                if z_reject is not None:
```

```
                    z_value_reject = (p_sample - p_nominal) /
np.sqrt(p_nominal * (1 - p_nominal) / n_samples)
                    if z_value_reject > z_reject:
                        reject_count += 1

                # 接收
                if z_accept is not None:
                    z_value_accept = (p_sample - p_nominal) /
np.sqrt(p_nominal * (1 - p_nominal) / n_samples)
                    if z_value_accept < z_accept:
                        accept_count += 1

            return reject_count, accept_count


        # 接受
        def find_sample_size_reject(p_nominal, z_reject,
min_n=10, max_n=200, step=1, num_trials=10000, tolerance=0.01,
                                    fluctuation=0.01):
            sample_sizes = []

            for n_samples in range(min_n, max_n, step):
                reject_count, _ = perform_test(n_samples,
p_nominal, z_reject=z_reject, z_accept=None,

num_trials=num_trials,

fluctuation=fluctuation)

                reject_rate = reject_count / num_trials

                print(f"样本量: {n_samples}, 拒收比例:
{reject_rate:.4f}")

                # 样本
                if abs(reject_rate - 0.05) <= tolerance:
                    sample_sizes.append(n_samples)

                # 10 个取均值
                if len(sample_sizes) >= 10:
                    avg_sample_size = np.mean(sample_sizes)
                    print(f"满足拒收条件的样本量为:
{sample_sizes}")
                    print(f"最优样本量的平均值为:
```

```
{avg_sample_size}")
                    return avg_sample_size

            print("未找到足够的样本量满足拒收条件。")
            return None


        # 接收
        def find_optimal_sample_size(p_nominal, z_accept,
min_n=10, max_n=200, step=1, num_trials=10000,
tolerance=0.01):
            sample_sizes = []

            for n_samples in range(min_n, max_n, step):
                _, accept_count = perform_test(n_samples,
p_nominal, z_accept=z_accept, num_trials=num_trials)

                accept_rate = accept_count / num_trials

                print(f"样本量: {n_samples}, 接收比例: {1-
accept_rate:.4f}")

                # 检查浮动
                if abs(accept_rate - 0.10) <= tolerance:
                    sample_sizes.append(n_samples)

                # 当找到 10 个符合条件的样本量时，取平均值
                if len(sample_sizes) >= 10:
                    avg_sample_size = np.mean(sample_sizes)
                    print(f"满足接收条件的样本量为:
{sample_sizes}")
                    print(f"最优样本量的平均值为:
{avg_sample_size}")
                    return avg_sample_size

            print("未找到足够的样本量满足接收条件。")
            return None


        optimal_sample_size_reject =
find_sample_size_reject(p_nominal, z_reject)
        optimal_sample_size_accept =
find_optimal_sample_size(p_nominal, z_accept)
```

问题二的求解

```
    import random
    import numpy as np
    from sklearn.tree import DecisionTreeRegressor
    import matplotlib.pyplot as plt
    from sklearn import tree

    # 定义零配件和成品的类
    class Part:
        def __init__(self, defect_rate, purchase_price,
detection_cost):
            self.defect_rate = defect_rate
            self.purchase_price = purchase_price
            self.detection_cost = detection_cost


    class FinalProduct:
        def __init__(self, defect_rate, assemble_cost,
detection_cost, market_price, exchange_loss,
disassemble_cost):
            self.defect_rate = defect_rate
            self.assemble_cost = assemble_cost
            self.detection_cost = detection_cost
            self.market_price = market_price
            self.exchange_loss = exchange_loss
            self.disassemble_cost = disassemble_cost


    # 引入误差的次品率
    def apply_defect_rate_with_error(defect_rate,
error_range=0.02):
        return defect_rate + random.uniform(-error_range,
error_range)


    # 计算总成本函数，加入误差
    def calculate_total_cost(case, detect_part1=True,
detect_part2=True, detect_final=True, disassemble=True,
error_range=0.02):
        part1 = case["零配件 1"]
        part2 = case["零配件 2"]
        final_product = case["成品"]
```

```
        # 应用误差后的次品率
        part1_defect =
apply_defect_rate_with_error(part1.defect_rate, error_range)
        part2_defect =
apply_defect_rate_with_error(part2.defect_rate, error_range)
        final_defect =
apply_defect_rate_with_error(final_product.defect_rate,
error_range)

        # 判断零配件是否合格
        part1_pass = random.random() > part1_defect
        part2_pass = random.random() > part2_defect

        # 如果有任何一个零配件不合格，成品必定不合格
        if not part1_pass or not part2_pass:
            final_pass = False
        else:
            # 如果两个零配件都合格，成品根据其次品率决定是否合格
            final_pass = random.random() > final_defect

        # 计算检测成本
        detection_cost = 0
        if detect_part1:
            detection_cost += part1.detection_cost
        if detect_part2:
            detection_cost += part2.detection_cost
        if detect_final:
            detection_cost += final_product.detection_cost

        # 计算装配成本
        assemble_cost = final_product.assemble_cost

        # 计算市场调换损失
        market_loss = 0
        if not final_pass and not detect_final:
            market_loss = final_product.exchange_loss * (1 -
final_pass)

        # 计算拆解费用或报废损失
        disassemble_cost = 0
        if not final_pass:
            if disassemble:
                disassemble_cost =
```

```
final_product.disassemble_cost
            else:
                scrap_cost = final_product.assemble_cost * (1 -
final_pass)
                disassemble_cost = scrap_cost

        total_cost = detection_cost + assemble_cost +
market_loss + disassemble_cost
        return total_cost


    # 生成数据集，用于训练决策树模型
    def generate_simulation_data(case, num_samples=1000,
error_range=0.02):
        X = []  # 特征集合
        y = []  # 成本集合

        for _ in range(num_samples):
            for detect_part1 in [True, False]:
                for detect_part2 in [True, False]:
                    for detect_final in [True, False]:
                        for disassemble in [True, False]:
                            # 生成数据集的特征和标签
                            total_cost =
calculate_total_cost(case, detect_part1, detect_part2,
detect_final, disassemble, error_range)
                            X.append([detect_part1, detect_part2,
detect_final, disassemble])
                            y.append(total_cost)

        return np.array(X), np.array(y)


    # 表 1 的六种情况
    cases = [
        {
            "零配件 1": Part(0.10, 4, 2),
            "零配件 2": Part(0.10, 18, 3),
            "成品": FinalProduct(0.10, 6, 3, 56, 6, 5)
        },
        {
            "零配件 1": Part(0.20, 4, 2),
            "零配件 2": Part(0.20, 18, 3),
            "成品": FinalProduct(0.20, 6, 3, 56, 6, 5)
```

```
        },
        {
            "零配件 1": Part(0.10, 4, 2),
            "零配件 2": Part(0.10, 18, 3),
            "成品": FinalProduct(0.10, 6, 3, 56, 30, 5)
        },
        {
            "零配件 1": Part(0.20, 4, 1),
            "零配件 2": Part(0.20, 18, 1),
            "成品": FinalProduct(0.20, 6, 2, 56, 30, 5)
        },
        {
            "零配件 1": Part(0.10, 4, 8),
            "零配件 2": Part(0.20, 18, 1),
            "成品": FinalProduct(0.10, 6, 2, 56, 10, 5)
        },
        {
            "零配件 1": Part(0.05, 4, 2),
            "零配件 2": Part(0.05, 18, 3),
            "成品": FinalProduct(0.05, 6, 3, 56, 10, 40)
        }
    ]

    # 使用决策树寻找每个 case 的最优解
    def optimize_case(case, case_index):
        # 生成模拟数据
        X, y = generate_simulation_data(case, num_samples=1000)

        # 使用决策树回归模型
        tree_reg = DecisionTreeRegressor(max_depth=5)
        tree_reg.fit(X, y)

        # 可视化决策树
        feature_names = ['Detect Part1', 'Detect Part2',
'Detect Final', 'Disassemble']
        visualize_decision_tree(tree_reg, feature_names,
case_index)

        # 模型预测：根据最小成本的组合预测决策
        predicted_costs = tree_reg.predict(X)

        # 找出最优决策
        min_cost_index = np.argmin(predicted_costs)
        optimal_decision = X[min_cost_index]
```

```
        return optimal_decision,
predicted_costs[min_cost_index]

    # 可视化决策树使用 matplotlib
    def visualize_decision_tree(tree_reg, feature_names,
case_index):
        plt.figure(figsize=(20,10))
        tree.plot_tree(tree_reg, feature_names=feature_names,
filled=True)
        plt.title(f'Decision Tree for Case {case_index}')
        plt.savefig(f"decision_tree_case_{case_index}.png")  #
保存为图片
        plt.show()


    # 遍历六种情况，输出每个情况的最优决策并可视化
    for i, case in enumerate(cases):
        optimal_decision, min_cost = optimize_case(case, i+1)
        print(f"情况 {i+1} 的最优决策: 零配件 1 检测:
{bool(optimal_decision[0])}, 零配件 2 检测:
{bool(optimal_decision[1])}, 成品检测:
{bool(optimal_decision[2])}, 拆解:
{bool(optimal_decision[3])}")
        print(f"对应的最小总成本: {min_cost:.2f}\n")


问题三 2 道工序 8 个零配件的求解
    import random
    import numpy as np
    from sklearn.tree import DecisionTreeRegressor
    import matplotlib.pyplot as plt
    from sklearn import tree
    from joblib import Parallel, delayed


    # 定义零配件和成品的类
    class Part:
        def __init__(self, defect_rate, purchase_price,
detection_cost):
            self.defect_rate = defect_rate
            self.purchase_price = purchase_price
            self.detection_cost = detection_cost
```

```
    class SemiProduct:
        def __init__(self, defect_rate, assemble_cost,
detection_cost, disassemble_cost):
            self.defect_rate = defect_rate
            self.assemble_cost = assemble_cost
            self.detection_cost = detection_cost
            self.disassemble_cost = disassemble_cost


    class FinalProduct:
        def __init__(self, defect_rate, assemble_cost,
detection_cost, market_price, exchange_loss,
disassemble_cost):
            self.defect_rate = defect_rate
            self.assemble_cost = assemble_cost
            self.detection_cost = detection_cost
            self.market_price = market_price
            self.exchange_loss = exchange_loss
            self.disassemble_cost = disassemble_cost


    # 引入误差的次品率
    def apply_defect_rate_with_error(defect_rate,
error_range=0.02):
        return defect_rate + random.uniform(-error_range,
error_range)


    # 计算全概率贝叶斯公式的概率
    def bayesian_final_defect_rate(semi_products,
final_product_defect_rate, error_range=0.02):
        total_defect_rate = 1
        for sp in semi_products:
            sp_defect_rate =
apply_defect_rate_with_error(sp.defect_rate, error_range)
            total_defect_rate *= (1 - sp_defect_rate)
        final_defect_rate =
apply_defect_rate_with_error(final_product_defect_rate,
error_range)
        total_defect_rate *= (1 - final_defect_rate)
        return 1 - total_defect_rate
```

```
    # 计算总成本函数，加入误差
    def calculate_total_cost(case, detect_part1=True,
detect_part2=True, detect_part3=True,
                             detect_part4=True,
detect_part5=True, detect_part6=True,
                             detect_part7=True,
detect_part8=True,
                             detect_semi1=True,
detect_semi2=True, detect_semi3=True,
                             detect_final=True, disassemble=True,
error_range=0.02):
        part1 = case["零配件 1"]
        part2 = case["零配件 2"]
        part3 = case["零配件 3"]
        part4 = case["零配件 4"]
        part5 = case["零配件 5"]
        part6 = case["零配件 6"]
        part7 = case["零配件 7"]
        part8 = case["零配件 8"]
        semi1 = case["半成品 1"]
        semi2 = case["半成品 2"]
        semi3 = case["半成品 3"]
        final_product = case["成品"]

        # 应用误差后的次品率
        part1_defect =
apply_defect_rate_with_error(part1.defect_rate, error_range)
        part2_defect =
apply_defect_rate_with_error(part2.defect_rate, error_range)
        part3_defect =
apply_defect_rate_with_error(part3.defect_rate, error_range)
        part4_defect =
apply_defect_rate_with_error(part4.defect_rate, error_range)
        part5_defect =
apply_defect_rate_with_error(part5.defect_rate, error_range)
        part6_defect =
apply_defect_rate_with_error(part6.defect_rate, error_range)
        part7_defect =
apply_defect_rate_with_error(part7.defect_rate, error_range)
        part8_defect =
apply_defect_rate_with_error(part8.defect_rate, error_range)

        # 半成品次品率的计算
        semi1_defect = 1 - ((1 - part1_defect) * (1 -
```

```
part2_defect) * (1 - part3_defect))
        semi2_defect = 1 - ((1 - part4_defect) * (1 -
part5_defect) * (1 - part6_defect))
        semi3_defect = 1 - ((1 - part7_defect) * (1 -
part8_defect))

        # 判断零配件是否合格
        semi1_pass = random.random() > semi1_defect
        semi2_pass = random.random() > semi2_defect
        semi3_pass = random.random() > semi3_defect

        # 如果有任何一个半成品不合格，成品必定不合格
        if not semi1_pass or not semi2_pass or not semi3_pass:
            final_pass = False
        else:
            # 如果半成品都合格，成品根据其次品率决定是否合格
            final_defect = bayesian_final_defect_rate([semi1,
semi2, semi3], final_product.defect_rate, error_range)
            final_pass = random.random() > final_defect

        # 计算检测成本
        detection_cost = 0
        if detect_part1:
            detection_cost += part1.detection_cost
        if detect_part2:
            detection_cost += part2.detection_cost
        if detect_part3:
            detection_cost += part3.detection_cost
        if detect_part4:
            detection_cost += part4.detection_cost
        if detect_part5:
            detection_cost += part5.detection_cost
        if detect_part6:
            detection_cost += part6.detection_cost
        if detect_part7:
            detection_cost += part7.detection_cost
        if detect_part8:
            detection_cost += part8.detection_cost
        if detect_semi1:
            detection_cost += semi1.detection_cost
        if detect_semi2:
            detection_cost += semi2.detection_cost
        if detect_semi3:
            detection_cost += semi3.detection_cost
```

```
        if detect_final:
            detection_cost += final_product.detection_cost

        # 计算装配成本
        assemble_cost = semi1.assemble_cost +
semi2.assemble_cost + semi3.assemble_cost +
final_product.assemble_cost

        # 计算市场调换损失
        market_loss = 0
        if not final_pass and not detect_final:
            market_loss = final_product.exchange_loss * (1 -
final_pass)

        # 计算拆解费用或报废损失
        disassemble_cost = 0
        if not final_pass:
            if disassemble:
                disassemble_cost =
final_product.disassemble_cost + semi1.disassemble_cost +
semi2.disassemble_cost + semi3.disassemble_cost
            else:
                scrap_cost = final_product.assemble_cost * (1 -
final_pass)
                disassemble_cost = scrap_cost

        total_cost = detection_cost + assemble_cost +
market_loss + disassemble_cost
        return total_cost


    # 生成数据集的函数（并行处理）
    def generate_simulation_data(case, num_samples=1000,
error_range=0.02):
        def generate_sample():
            X = []
            y = []
            for detect_part1 in [True, False]:
                for detect_part2 in [True, False]:
                    for detect_part3 in [True, False]:
                        for detect_part4 in [True, False]:
                            for detect_part5 in [True, False]:
                                for detect_part6 in [True,
False]:
```

```
                                    for detect_part7 in [True,
False]:
                                        for detect_part8 in [True,
False]:
                                            for detect_semi1 in
[True, False]:
                                                for detect_semi2 in
[True, False]:
                                                    for
detect_semi3 in [True, False]:
                                                        for
detect_final in [True, False]:
                                                            for
disassemble in [True, False]:
total_cost = calculate_total_cost(
case, detect_part1, detect_part2, detect_part3,
detect_part4, detect_part5, detect_part6,
detect_part7, detect_part8,
detect_semi1, detect_semi2, detect_semi3,
detect_final, disassemble, error_range)
X.append([detect_part1, detect_part2, detect_part3,
detect_part4, detect_part5, detect_part6,
detect_part7, detect_part8,
detect_semi1, detect_semi2, detect_semi3,
detect_final, disassemble])
y.append(total_cost)
           return np.array(X), np.array(y)

       # 使用 joblib 并行生成数据
       results = Parallel(n_jobs=-
1)(delayed(generate_sample)() for _ in range(num_samples))
```

```
    # 合并结果
    X = np.vstack([r[0] for r in results])
    y = np.hstack([r[1] for r in results])

    return X, y


# 表 2 的生产情况
case3 = {
    "零配件 1": Part(0.10, 2, 1),
    "零配件 2": Part(0.10, 8, 1),
    "零配件 3": Part(0.10, 12, 2),
    "零配件 4": Part(0.10, 2, 1),
    "零配件 5": Part(0.10, 6, 1),
    "零配件 6": Part(0.10, 8, 1),
    "零配件 7": Part(0.10, 6, 1),
    "零配件 8": Part(0.10, 2, 1),
    "半成品 1": SemiProduct(0.10, 3, 2, 5),
    "半成品 2": SemiProduct(0.10, 3, 1, 2),
    "半成品 3": SemiProduct(0.10, 3, 2, 5),
    "成品": FinalProduct(0.10, 3, 4, 50, 10, 10),
}

# 生成数据
X, y = generate_simulation_data(case3)

# 使用决策树训练模型
model = DecisionTreeRegressor()
model.fit(X, y)

# 可视化决策树
plt.figure(figsize=(20, 10))
tree.plot_tree(model, filled=True, feature_names=[
    'Detect Part 1', 'Detect Part 2', 'Detect Part 3',
    'Detect Part 4', 'Detect Part 5', 'Detect Part 6',
    'Detect Part 7', 'Detect Part 8',
    'Detect Semi 1', 'Detect Semi 2', 'Detect Semi 3',
    'Detect Final', 'Disassemble'])
plt.show()
```

问题三 m 道工序 n 个零配件的求解

```
import random
import numpy as np
```

```
    # 定义零配件、中间成品和成品的类
    class Part:
        def __init__(self, defect_rate, purchase_price,
detection_cost):
            self.defect_rate = defect_rate
            self.purchase_price = purchase_price
            self.detection_cost = detection_cost

    class SemiProduct:
        def __init__(self, defect_rate, assemble_cost,
detection_cost, disassemble_cost):
            self.defect_rate = defect_rate
            self.assemble_cost = assemble_cost
            self.detection_cost = detection_cost
            self.disassemble_cost = disassemble_cost

    class FinalProduct:
        def __init__(self, defect_rate, assemble_cost,
detection_cost, disassemble_cost, market_price,
exchange_loss):
            self.defect_rate = defect_rate
            self.assemble_cost = assemble_cost
            self.detection_cost = detection_cost
            self.disassemble_cost = disassemble_cost
            self.market_price = market_price
            self.exchange_loss = exchange_loss

    # 计算总成本函数
    def calculate_total_cost(parts, semi_products,
final_product, detect_flags):

        # 计算零配件的检测通过率
        part_passes = [random.random() > part.defect_rate for
part in parts]

        # 根据零配件检测通过率计算中间成品的通过率
        semi_passes = []
        parts_per_semi = len(parts) // len(semi_products)
        for i, semi in enumerate(semi_products):
            start = i * parts_per_semi
            end = start + parts_per_semi
            semi_passes.append(all(part_passes[start:end]) and
(random.random() > semi.defect_rate))
```

```
        # 计算成品的通过率
        final_pass = all(semi_passes) and (random.random() >
final_product.defect_rate)

        # 计算检测成本
        detection_cost = sum(part.detection_cost for part, flag
in zip(parts, detect_flags[:len(parts)]) if flag)
        detection_cost += sum(semi.detection_cost for semi,
flag in zip(semi_products,
detect_flags[len(parts):len(parts)+len(semi_products)]) if
flag)
        if detect_flags[-2]:
            detection_cost += final_product.detection_cost

        # 计算装配成本
        assemble_cost = sum(semi.assemble_cost for semi in
semi_products) + final_product.assemble_cost

        # 计算市场调换损失
        market_loss = 0
        if not final_pass and not detect_flags[-2]:
            market_loss = final_product.exchange_loss

        # 计算拆解费用或报废损失
        disassemble_cost = 0
        if not final_pass:
            if detect_flags[-1]:
                disassemble_cost =
final_product.disassemble_cost
            else:
                scrap_cost = final_product.assemble_cost
                disassemble_cost = scrap_cost

        total_cost = detection_cost + assemble_cost +
market_loss + disassemble_cost
        return total_cost

    # 生成零配件、中间成品和成品的实例
    def generate_case(m, n):
        parts = [Part(0.10, 8, 1.375) for _ in range(n)]
        semi_products = [SemiProduct(0.10, 8, 4, 6) for _ in
range(m)]
        final_product = FinalProduct(0.10, 8, 4, 6, 25 * n, 5 *
```

```
n)
        return parts, semi_products, final_product

    # 模拟生产过程
    def simulate_production(m, n, simulations=1000):
        parts, semi_products, final_product = generate_case(m,
n)
        detect_flags = [True] * (n + m + 2)  # 所有的检测和拆解标
志都设置为 True

        total_costs = []
        for _ in range(simulations):
            total_cost = calculate_total_cost(parts,
semi_products, final_product, detect_flags)
            total_costs.append(total_cost)

        average_cost = np.mean(total_costs)
        return average_cost

    # 输入 m 和 n
    m = int(input("请输入中间成品的数量（m）："))
    n = int(input("请输入零配件的数量（n）："))

    # 运行模拟
    average_cost = simulate_production(m, n)
    print(f"对于{m}道工序、{n}个零配件的平均总成本为：
{average_cost:.2f}")
```

问题四 p 区间的确定

```
    import numpy as np
    import scipy.stats as stats

    # 样本量和接收比例数据
    sample_sizes = np.array([
        10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25,
        26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
40, 41,
        42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55,
56, 57,
        58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71,
72, 73,
        74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87,
```

```
88, 89,
       90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102,
103, 104,
       105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115,
116, 117,
       118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128,
129, 130,
       131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141,
142, 143,
       144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154,
155, 156,
       157, 158, 159, 160, 161, 162, 163
    ])

    acceptance_ratios = np.array([
       1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.7959, 0.8116,
0.8261, 0.8506,
       0.8684, 0.8791, 0.8913, 0.9039, 0.9095, 0.9153, 0.9251,
0.9366, 0.9364,
       0.9457, 0.9478, 0.9565, 0.9605, 0.8406, 0.8562, 0.8725,
0.8785, 0.8828,
       0.8938, 0.9050, 0.9126, 0.9208, 0.9264, 0.9328, 0.9370,
0.9427, 0.9457,
       0.9480, 0.8532, 0.8701, 0.8802, 0.8828, 0.8933, 0.8981,
0.9048, 0.9151,
       0.9160, 0.9307, 0.9344, 0.9371, 0.9378, 0.8564, 0.8661,
0.8732, 0.8806,
       0.8887, 0.8971, 0.9049, 0.9111, 0.9177, 0.9245, 0.9240,
0.9336, 0.9329,
       0.8626, 0.8750, 0.8806, 0.8891, 0.8881, 0.8953, 0.9002,
0.9107, 0.9143,
       0.9200, 0.9190, 0.9318, 0.9308, 0.8640, 0.8797, 0.8846,
0.8845, 0.8911,
       0.8981, 0.9089, 0.9068, 0.9144, 0.9200, 0.9226, 0.9300,
0.9336, 0.8734,
       0.8739, 0.8864, 0.8911, 0.8953, 0.9013, 0.9086, 0.9100,
0.9105, 0.9188,
       0.9239, 0.9302, 0.8729, 0.8779, 0.8835, 0.8859, 0.8905,
0.8995, 0.9027,
       0.9075, 0.9109, 0.9142, 0.9193, 0.9232, 0.8739, 0.8729,
0.8825, 0.8890,
       0.8922, 0.8977, 0.8986, 0.9067, 0.9089, 0.9135, 0.9230,
0.9262, 0.8605,
       0.8817, 0.8817, 0.8809, 0.8921, 0.8967, 0.8982, 0.9092,
```

```
0.9117, 0.9109,
       0.9226, 0.9227, 0.8723, 0.8836, 0.8815, 0.8865, 0.8901,
0.8978, 0.9016,
       0.9079, 0.9076, 0.9085, 0.9124, 0.9189, 0.8747, 0.8809,
0.8844, 0.8857,
       0.8938
    ])

    # z-score for 99% confidence
    z = stats.norm.ppf(1 - 0.01/2)

    # Calculate the confidence intervals
    confidence_intervals = []
    for n, p_hat in zip(sample_sizes, acceptance_ratios):
        se = np.sqrt(p_hat * (1 - p_hat) / n)
        margin_of_error = z * se
        confidence_interval = (p_hat - margin_of_error, p_hat +
margin_of_error)
        confidence_intervals.append(confidence_interval)

    # Find the overall interval covering all the confidence
intervals
    lower_bound = max([ci[0] for ci in confidence_intervals])
    upper_bound = min([ci[1] for ci in confidence_intervals])

    print(1-upper_bound)
```

问题四下问题 2 的求解

```
    import random
    import numpy as np
    from sklearn.tree import DecisionTreeRegressor
    import matplotlib.pyplot as plt
    from sklearn import tree

    # 定义零配件和成品的类
    class Part:
        def __init__(self, defect_rate, purchase_price,
detection_cost):
            self.defect_rate = defect_rate
            self.purchase_price = purchase_price
            self.detection_cost = detection_cost


    class FinalProduct:
```

```
        def __init__(self, defect_rate, assemble_cost,
detection_cost, market_price, exchange_loss,
disassemble_cost):
            self.defect_rate = defect_rate
            self.assemble_cost = assemble_cost
            self.detection_cost = detection_cost
            self.market_price = market_price
            self.exchange_loss = exchange_loss
            self.disassemble_cost = disassemble_cost


    # 引入误差的次品率，通过正态分布模拟
    def apply_defect_rate_with_error(defect_rate,
error_range=0.06269088936839551):
        # 使用正态分布进行浮动模拟
        return np.random.normal(defect_rate, error_range / 2)



    # 计算总成本函数，加入误差
    def calculate_total_cost(case, detect_part1=True,
detect_part2=True, detect_final=True, disassemble=True,
error_range=0.02):
        part1 = case["零配件 1"]
        part2 = case["零配件 2"]
        final_product = case["成品"]

        # 应用误差后的次品率
        part1_defect =
apply_defect_rate_with_error(part1.defect_rate, error_range)
        part2_defect =
apply_defect_rate_with_error(part2.defect_rate, error_range)
        final_defect =
apply_defect_rate_with_error(final_product.defect_rate,
error_range)

        # 判断零配件是否合格
        part1_pass = random.random() > part1_defect
        part2_pass = random.random() > part2_defect

        # 如果有任何一个零配件不合格，成品必定不合格
        if not part1_pass or not part2_pass:
            final_pass = False
        else:
```

```
            # 如果两个零配件都合格，成品根据其次品率决定是否合格
            final_pass = random.random() > final_defect

        # 计算检测成本
        detection_cost = 0
        if detect_part1:
            detection_cost += part1.detection_cost
        if detect_part2:
            detection_cost += part2.detection_cost
        if detect_final:
            detection_cost += final_product.detection_cost

        # 计算装配成本
        assemble_cost = final_product.assemble_cost

        # 计算市场调换损失
        market_loss = 0
        if not final_pass and not detect_final:
            market_loss = final_product.exchange_loss * (1 -
final_pass)

        # 计算拆解费用或报废损失
        disassemble_cost = 0
        if not final_pass:
            if disassemble:
                disassemble_cost =
final_product.disassemble_cost
            else:
                scrap_cost = final_product.assemble_cost * (1 -
final_pass)
                disassemble_cost = scrap_cost

        total_cost = detection_cost + assemble_cost +
market_loss + disassemble_cost
        return total_cost


    # 生成数据集，用于训练决策树模型
    def generate_simulation_data(case, num_samples=1000,
error_range=0.02):
        X = []  # 特征集合
        y = []  # 成本集合

        for _ in range(num_samples):
```

```
            for detect_part1 in [True, False]:
                for detect_part2 in [True, False]:
                    for detect_final in [True, False]:
                        for disassemble in [True, False]:
                            # 生成数据集的特征和标签
                            total_cost =
calculate_total_cost(case, detect_part1, detect_part2,
detect_final, disassemble, error_range)
                            X.append([detect_part1, detect_part2,
detect_final, disassemble])
                            y.append(total_cost)

        return np.array(X), np.array(y)


    # 表 1 的六种情况
    cases = [
        {
            "零配件 1": Part(0.10, 4, 2),
            "零配件 2": Part(0.10, 18, 3),
            "成品": FinalProduct(0.10, 6, 3, 56, 6, 5)
        },
        {
            "零配件 1": Part(0.20, 4, 2),
            "零配件 2": Part(0.20, 18, 3),
            "成品": FinalProduct(0.20, 6, 3, 56, 6, 5)
        },
        {
            "零配件 1": Part(0.10, 4, 2),
            "零配件 2": Part(0.10, 18, 3),
            "成品": FinalProduct(0.10, 6, 3, 56, 30, 5)
        },
        {
            "零配件 1": Part(0.20, 4, 1),
            "零配件 2": Part(0.20, 18, 1),
            "成品": FinalProduct(0.20, 6, 2, 56, 30, 5)
        },
        {
            "零配件 1": Part(0.10, 4, 8),
            "零配件 2": Part(0.20, 18, 1),
            "成品": FinalProduct(0.10, 6, 2, 56, 10, 5)
        },
        {
            "零配件 1": Part(0.05, 4, 2),
```

```
            "零配件 2": Part(0.05, 18, 3),
            "成品": FinalProduct(0.05, 6, 3, 56, 10, 40)
        }
    ]

    # 使用决策树寻找每个 case 的最优解
    def optimize_case(case, case_index):
        # 生成模拟数据
        X, y = generate_simulation_data(case, num_samples=1000)

        # 使用决策树回归模型
        tree_reg = DecisionTreeRegressor(max_depth=5)
        tree_reg.fit(X, y)

        # 可视化决策树
        feature_names = ['Detect Part1', 'Detect Part2',
'Detect Final', 'Disassemble']
        visualize_decision_tree(tree_reg, feature_names,
case_index)

        # 模型预测：根据最小成本的组合预测决策
        predicted_costs = tree_reg.predict(X)

        # 找出最优决策
        min_cost_index = np.argmin(predicted_costs)
        optimal_decision = X[min_cost_index]

        return optimal_decision,
predicted_costs[min_cost_index]

    # 可视化决策树使用 matplotlib
    def visualize_decision_tree(tree_reg, feature_names,
case_index):
        plt.figure(figsize=(20,10))
        tree.plot_tree(tree_reg, feature_names=feature_names,
filled=True)
        plt.title(f'Decision Tree for Case {case_index}')
        plt.savefig(f"decision_tree_case_{case_index}.png")  #
保存为图片
        plt.show()


    # 遍历六种情况，输出每个情况的最优决策并可视化
    for i, case in enumerate(cases):
```

```
        optimal_decision, min_cost = optimize_case(case, i+1)
        print(f"情况 {i+1} 的最优决策: 零配件 1 检测:
{bool(optimal_decision[0])}, 零配件 2 检测:
{bool(optimal_decision[1])}, 成品检测:
{bool(optimal_decision[2])}, 拆解:
{bool(optimal_decision[3])}")
        print(f"对应的最小总成本: {min_cost:.2f}\n")
```

问题四下问题 3 的 2 道工序 8 个零配件的求解

```
    import random
    import numpy as np
    from sklearn.tree import DecisionTreeRegressor
    import matplotlib.pyplot as plt
    from sklearn import tree
    from joblib import Parallel, delayed


    # 定义零配件和成品的类
    class Part:
        def __init__(self, defect_rate, purchase_price,
detection_cost):
            self.defect_rate = defect_rate
            self.purchase_price = purchase_price
            self.detection_cost = detection_cost


    class SemiProduct:
        def __init__(self, defect_rate, assemble_cost,
detection_cost, disassemble_cost):
            self.defect_rate = defect_rate
            self.assemble_cost = assemble_cost
            self.detection_cost = detection_cost
            self.disassemble_cost = disassemble_cost


    class FinalProduct:
        def __init__(self, defect_rate, assemble_cost,
detection_cost, market_price, exchange_loss,
disassemble_cost):
            self.defect_rate = defect_rate
            self.assemble_cost = assemble_cost
            self.detection_cost = detection_cost
            self.market_price = market_price
```

```
            self.exchange_loss = exchange_loss
            self.disassemble_cost = disassemble_cost


    # 引入误差的次品率，通过正态分布模拟
    def apply_defect_rate_with_error(defect_rate,
error_range=0.06269088936839551):
        # 使用正态分布进行浮动模拟
        return np.random.normal(defect_rate, error_range / 2)


    # 计算全概率贝叶斯公式的概率
    def bayesian_final_defect_rate(semi_products,
final_product_defect_rate, error_range=0.02):
        total_defect_rate = 1
        for sp in semi_products:
            sp_defect_rate =
apply_defect_rate_with_error(sp.defect_rate, error_range)
            total_defect_rate *= (1 - sp_defect_rate)
        final_defect_rate =
apply_defect_rate_with_error(final_product_defect_rate,
error_range)
        total_defect_rate *= (1 - final_defect_rate)
        return 1 - total_defect_rate


    # 计算总成本函数，加入误差
    def calculate_total_cost(case, detect_part1=True,
detect_part2=True, detect_part3=True,
                             detect_part4=True,
detect_part5=True, detect_part6=True,
                             detect_part7=True,
detect_part8=True,
                             detect_semi1=True,
detect_semi2=True, detect_semi3=True,
                             detect_final=True, disassemble=True,
error_range=0.02):
        part1 = case["零配件 1"]
        part2 = case["零配件 2"]
        part3 = case["零配件 3"]
        part4 = case["零配件 4"]
        part5 = case["零配件 5"]
        part6 = case["零配件 6"]
        part7 = case["零配件 7"]
```

```
        part8 = case["零配件 8"]
        semi1 = case["半成品 1"]
        semi2 = case["半成品 2"]
        semi3 = case["半成品 3"]
        final_product = case["成品"]

        # 应用误差后的次品率
        part1_defect =
apply_defect_rate_with_error(part1.defect_rate, error_range)
        part2_defect =
apply_defect_rate_with_error(part2.defect_rate, error_range)
        part3_defect =
apply_defect_rate_with_error(part3.defect_rate, error_range)
        part4_defect =
apply_defect_rate_with_error(part4.defect_rate, error_range)
        part5_defect =
apply_defect_rate_with_error(part5.defect_rate, error_range)
        part6_defect =
apply_defect_rate_with_error(part6.defect_rate, error_range)
        part7_defect =
apply_defect_rate_with_error(part7.defect_rate, error_range)
        part8_defect =
apply_defect_rate_with_error(part8.defect_rate, error_range)

        # 半成品次品率的计算
        semi1_defect = 1 - ((1 - part1_defect) * (1 -
part2_defect) * (1 - part3_defect))
        semi2_defect = 1 - ((1 - part4_defect) * (1 -
part5_defect) * (1 - part6_defect))
        semi3_defect = 1 - ((1 - part7_defect) * (1 -
part8_defect))

        # 判断零配件是否合格
        semi1_pass = random.random() > semi1_defect
        semi2_pass = random.random() > semi2_defect
        semi3_pass = random.random() > semi3_defect

        # 如果有任何一个半成品不合格，成品必定不合格
        if not semi1_pass or not semi2_pass or not semi3_pass:
            final_pass = False
        else:
            # 如果半成品都合格，成品根据其次品率决定是否合格
            final_defect = bayesian_final_defect_rate([semi1,
semi2, semi3], final_product.defect_rate, error_range)
```

```
            final_pass = random.random() > final_defect

        # 计算检测成本
        detection_cost = 0
        if detect_part1:
            detection_cost += part1.detection_cost
        if detect_part2:
            detection_cost += part2.detection_cost
        if detect_part3:
            detection_cost += part3.detection_cost
        if detect_part4:
            detection_cost += part4.detection_cost
        if detect_part5:
            detection_cost += part5.detection_cost
        if detect_part6:
            detection_cost += part6.detection_cost
        if detect_part7:
            detection_cost += part7.detection_cost
        if detect_part8:
            detection_cost += part8.detection_cost
        if detect_semi1:
            detection_cost += semi1.detection_cost
        if detect_semi2:
            detection_cost += semi2.detection_cost
        if detect_semi3:
            detection_cost += semi3.detection_cost
        if detect_final:
            detection_cost += final_product.detection_cost

        # 计算装配成本
        assemble_cost = semi1.assemble_cost +
semi2.assemble_cost + semi3.assemble_cost +
final_product.assemble_cost

        # 计算市场调换损失
        market_loss = 0
        if not final_pass and not detect_final:
            market_loss = final_product.exchange_loss * (1 -
final_pass)

        # 计算拆解费用或报废损失
        disassemble_cost = 0
        if not final_pass:
            if disassemble:
```

```
                disassemble_cost =
final_product.disassemble_cost + semi1.disassemble_cost +
semi2.disassemble_cost + semi3.disassemble_cost
            else:
                scrap_cost = final_product.assemble_cost * (1 -
final_pass)
                disassemble_cost = scrap_cost

        total_cost = detection_cost + assemble_cost +
market_loss + disassemble_cost
        return total_cost


    # 生成数据集的函数（并行处理）
    def generate_simulation_data(case, num_samples=1000,
error_range=0.02):
        def generate_sample():
            X = []
            y = []
            for detect_part1 in [True, False]:
                for detect_part2 in [True, False]:
                    for detect_part3 in [True, False]:
                        for detect_part4 in [True, False]:
                            for detect_part5 in [True, False]:
                                for detect_part6 in [True,
False]:
                                    for detect_part7 in [True,
False]:
                                        for detect_part8 in [True,
False]:
                                            for detect_semi1 in
[True, False]:
                                                for detect_semi2 in
[True, False]:
                                                    for
detect_semi3 in [True, False]:
                                                        for
detect_final in [True, False]:
                                                            for
disassemble in [True, False]:

total_cost = calculate_total_cost(

case, detect_part1, detect_part2, detect_part3,
```

```
detect_part4, detect_part5, detect_part6,

detect_part7, detect_part8,

detect_semi1, detect_semi2, detect_semi3,

detect_final, disassemble, error_range)

X.append([detect_part1, detect_part2, detect_part3,

detect_part4, detect_part5, detect_part6,

detect_part7, detect_part8,

detect_semi1, detect_semi2, detect_semi3,

detect_final, disassemble])

y.append(total_cost)
           return np.array(X), np.array(y)

        # 使用 joblib 并行生成数据
        results = Parallel(n_jobs=-
1)(delayed(generate_sample)() for _ in range(num_samples))

        # 合并结果
        X = np.vstack([r[0] for r in results])
        y = np.hstack([r[1] for r in results])

        return X, y


    # 表 2 的生产情况
    case3 = {
        "零配件 1": Part(0.10, 2, 1),
        "零配件 2": Part(0.10, 8, 1),
        "零配件 3": Part(0.10, 12, 2),
        "零配件 4": Part(0.10, 2, 1),
        "零配件 5": Part(0.10, 6, 1),
        "零配件 6": Part(0.10, 8, 1),
        "零配件 7": Part(0.10, 6, 1),
        "零配件 8": Part(0.10, 2, 1),
        "半成品 1": SemiProduct(0.10, 3, 2, 5),
```

```
        "半成品 2": SemiProduct(0.10, 3, 1, 2),
        "半成品 3": SemiProduct(0.10, 3, 2, 5),
        "成品": FinalProduct(0.10, 3, 4, 50, 10, 10),
    }

    # 生成数据
    X, y = generate_simulation_data(case3)

    # 使用决策树训练模型
    model = DecisionTreeRegressor()
    model.fit(X, y)

    # 可视化决策树
    plt.figure(figsize=(20, 10))
    tree.plot_tree(model, filled=True, feature_names=[
        'Detect Part 1', 'Detect Part 2', 'Detect Part 3',
        'Detect Part 4', 'Detect Part 5', 'Detect Part 6',
        'Detect Part 7', 'Detect Part 8',
        'Detect Semi 1', 'Detect Semi 2', 'Detect Semi 3',
        'Detect Final', 'Disassemble'])
    plt.show()
```

问题四下问题 3 的 m 个配件 n 道工序的求解

```
    import random
    import numpy as np

    # 定义零配件、中间成品和成品的类
    class Part:
        def __init__(self, defect_rate, purchase_price,
detection_cost):
            self.defect_rate = defect_rate
            self.purchase_price = purchase_price
            self.detection_cost = detection_cost

    class SemiProduct:
        def __init__(self, defect_rate, assemble_cost,
detection_cost, disassemble_cost):
            self.defect_rate = defect_rate
            self.assemble_cost = assemble_cost
            self.detection_cost = detection_cost
            self.disassemble_cost = disassemble_cost

    class FinalProduct:
```

```
        def __init__(self, defect_rate, assemble_cost,
detection_cost, disassemble_cost, market_price,
exchange_loss):
            self.defect_rate = defect_rate
            self.assemble_cost = assemble_cost
            self.detection_cost = detection_cost
            self.disassemble_cost = disassemble_cost
            self.market_price = market_price
            self.exchange_loss = exchange_loss

    # 计算总成本函数
    def calculate_total_cost(parts, semi_products,
final_product, detect_flags):

        # 计算零配件的检测通过率
        part_passes = [random.random() > part.defect_rate for
part in parts]

        # 根据零配件检测通过率计算中间成品的通过率
        semi_passes = []
        parts_per_semi = len(parts) // len(semi_products)
        for i, semi in enumerate(semi_products):
            start = i * parts_per_semi
            end = start + parts_per_semi
            semi_passes.append(all(part_passes[start:end]) and
(random.random() > semi.defect_rate))

        # 计算成品的通过率
        final_pass = all(semi_passes) and (random.random() >
final_product.defect_rate)

        # 计算检测成本
        detection_cost = sum(part.detection_cost for part, flag
in zip(parts, detect_flags[:len(parts)]) if flag)
        detection_cost += sum(semi.detection_cost for semi,
flag in zip(semi_products,
detect_flags[len(parts):len(parts)+len(semi_products)]) if
flag)
        if detect_flags[-2]:
            detection_cost += final_product.detection_cost

        # 计算装配成本
        assemble_cost = sum(semi.assemble_cost for semi in
semi_products) + final_product.assemble_cost
```

```
        # 计算市场调换损失
        market_loss = 0
        if not final_pass and not detect_flags[-2]:
            market_loss = final_product.exchange_loss

        # 计算拆解费用或报废损失
        disassemble_cost = 0
        if not final_pass:
            if detect_flags[-1]:
                disassemble_cost =
final_product.disassemble_cost
            else:
                scrap_cost = final_product.assemble_cost
                disassemble_cost = scrap_cost

        total_cost = detection_cost + assemble_cost +
market_loss + disassemble_cost
        return total_cost

    # 生成零配件、中间成品和成品的实例
    def generate_case(m, n):
        parts = [Part(0.10, 8, 1.375) for _ in range(n)]
        semi_products = [SemiProduct(0.10, 8, 4, 6) for _ in
range(m)]
        final_product = FinalProduct(0.10, 8, 4, 6, 25 * n, 5 *
n)
        return parts, semi_products, final_product

    # 模拟生产过程
    def simulate_production(m, n, simulations=1000):
        parts, semi_products, final_product = generate_case(m,
n)
        detect_flags = [True] * (n + m + 2)  # 所有的检测和拆解标
志都设置为 True

        total_costs = []
        for _ in range(simulations):
            total_cost = calculate_total_cost(parts,
semi_products, final_product, detect_flags)
            total_costs.append(total_cost)

        average_cost = np.mean(total_costs)
        return average_cost
```

```
for i in range(1,11):
    m=i
    for j in range(1,11):
        n=j
        average_cost = simulate_production(m, n)
        print(f"{m} {n} {average_cost:.2f}")
```