多模型融合与贝叶斯网络驱动的慢性病预测及共病分析

摘要

慢性疾病已成为全球公共卫生领域的核心挑战，其早期预测与多病关联分析对精准防控具有重大意义。本文聚焦心脏病、中风、肝硬化三类高发慢性疾病，通过“数据深度治理 - 精准预测建模 - 共病机制解析 - 防控策略转化”的全链条研究，构建了科学系统的解决方案。

针对问题一，即数据治理层面，本文系统整合 6446 例异构医疗数据，针对肝硬化数据集 32.06% 的胆固醇缺失等复杂情况，创新采用**中位数填补**（连续变量）与**众数填补**（分类变量）策略实现高质量预处理。通过 **U 检验**和**卡方检验**挖掘出关键临床规律：中风患者年龄跨度显著（0.08-82 岁），高血压暴露率（13.3%）与心脏病史（5.4%）呈强关联；心脏病患者中男性占比高达 79.0%，ST-T 波异常斜率效应量达 1.12；肝硬化患者胆红素水平为健康组的 3.3 倍，腹水患者死亡率更是高达 95%。

针对问题二，即预测模型构建方面，基于 **SHAP 特征重要性**排序筛选关键变量，为三类疾病量身打造专属预测架构：采用 **MLP 神经网络**精准捕捉肝硬化患者胆红素与血小板的非线性交互关系，模型 AUC 值达 0.9951；利用 **SVM 高斯核**有效映射心脏病 ST-T 波异常特征，AUC 值为 0.9443；创新设计 **XGBoost 加权损失函数**成功解决中风数据极端不平衡问题（阳性率 ＜5%），AUC 值达到 0.9951。经特征扰动测试验证，模型具有强鲁棒性（年龄 ±5% 时 AUC 波动 ＜0.018）。

针对问题三，在共病机制解析上，通过**贝叶斯网络**量化关键发现：“中风 + 肝硬化”共病概率达 0.018768，是“三病共存”概率的 1.81 倍；胆固醇作为核心枢纽因子，每升高 1mmol/L 共病风险增加 65%；老年组（≥60 岁）共病风险为青年组的 2.63 倍。据此构建的年龄分层干预模型，形成了以胆固醇控制（青年组）、肝功监测（中年组）、综合管理（老年组）为核心的分级防控体系。

针对问题四，旨在将心脏病、中风和肝硬化三种疾病的前期研究成果转化为具有针对性、科学性和可行性的公共卫生预防策略。通过对单疾病关键风险因素和共病特征的深度整合，构建了多维度预防模型框架。该框架强调分层干预和政策保障，目标是通过降低发病率和共病率，改善人群健康水平并减轻全球公共卫生负担。

本文创新采用“组合预测模型 + 多病网络”双驱动架构，模型经严格验证显示出优异性能与强鲁棒性，可扩展至糖尿病、肾病等慢性病联合防控体系，为临床干预提供了坚实的量化决策支持，对提升慢性病防控水平具有重要的理论与实践价值。

关键词：Spearman 相关系数、XGBoost、SVM、MLP、贝叶斯网络

一、 问题重述

随着全球人口老龄化和生活方式的变化，心血管疾病、中风和肝硬化等慢性疾病的发病率持续上升，已成为影响人类健康的主要威胁。世界卫生组织的数据显示，心血管疾病是全球死亡的首要原因，而中风和肝硬化同样导致极高的致残率和死亡率。这些疾病的发生往往与多种危险因素相关，如高血压、高血糖、肥胖、吸烟等，且不同疾病之间可能存在潜在的关联性。因此，通过大数据分析技术挖掘疾病风险因素、建立预测模型并探索多疾病间的关联规律，对早期干预和公共卫生管理具有重要意义。

【问题一】题目给定心脏病、中风和肝硬化三个数据集，要求对数据进行清洗、缺失值处理和异常值检测，并通过探索性分析揭示各疾病的关键统计特征和分布规律。

【问题二】基于预处理后的数据，需要分别建立心脏病、中风和肝硬化的预测模型，要求模型能够准确识别高风险个体，并分析不同特征对疾病发生的影响程度。

【问题三】在单疾病预测的基础上，进一步探索三种疾病之间的关联性，例如是否存在共同的危险因素或共病模式，并量化分析不同疾病组合发生的概率。

【问题四】结合模型分析结果，向世界卫生组织提交一份建议报告，提出针对高风险人群的早期筛查策略和疾病预防措施，为公共卫生决策提供数据支持。

二、 问题分析

2.1 问题一的分析

针对心脏病、中风、肝硬化三类异构医疗数据集，核心挑战在于处理数据质量问题并挖掘疾病特异性分布规律。通过数据清洗和探索性分析，需揭示各疾病的统计特征：中风患者年龄跨度极大（0.08-82 岁）且与高血压强相关；心脏病呈现男性主导（79.0%）和 ST-T 波异常高发特征；肝硬化则表现为胆红素水平异常升高（5.27 mg/dL）和腹水与高死亡率的关联。分析需通过统计检验(U 检验/卡方检验)量化风险因素效应量(如年龄对中风的效应量 1.14)，并利用可视化工具（如分布直方图、箱线图）直观展示关键指标的离散性与集中趋势，为后续建模奠定数据基础。

2.2 问题二的分析

针对三类疾病的不同数据特性，需设计差异化预测框架。对肝硬化采用 MLP 神经网络以捕捉非线性交互；对心脏病采用 SVM 高斯核映射高维特征边界；对中风采用 XGBoost 加权损失函数提升少数类识别能力。需通过网格搜索优化超参数（如 SVM 的惩罚系数 C=3.2、核系数 γ=0.08），并建立动态阈值机制（如肝硬化决策阈值从 0.5 提升至 0.65）平衡敏感性与特异性。

2.3 问题三的分析

基于单疾病预测结果，需探索心脏病、中风、肝硬化的共病机制。通过互信息检验（$I(X;Y) > 0.35$保留边）确定疾病节点间的因果关系，并计算条件概率。重点分析代谢综合征（胆固醇+BMI+血糖）作为共病基础的通路机制，结合年龄分层（青年/中年/老年组）揭示共病风险梯度(老年组风险为青年组的 2.63 倍)。最终通过有向无环图可视化多病关联网络，

为分级防控提供理论依据。

2.4 问题四的分析

本问题立足于心脏病、中风和肝硬化三大疾病对全球公共卫生造成的复合负担，旨在将前三题的研究成果，包括问题一揭示的单疾病关键风险因素、问题二构建的高精度预测模型，以及问题三挖掘的共病机制，转化为系统性预防策略。由于共病风险存在显著年龄分层且当前预防体系缺乏对代谢综合征共病本质的整合干预，亟需以单疾病风险特征支撑精准预防，以共病枢纽因子驱动源头管控，以预测模型赋能高风险定位，最终通过政策保障模块解决数据碎片化与资源不均衡的落地瓶颈，实现从科研实证到公共卫生实践的跃迁。

三、 模型假设

（1）题目提供的数据集中的测量值准确可靠，缺失值和异常值已在预处理阶段合理处理，不影响模型的整体有效性。

（2）在单疾病预测模型中，假设各特征变量之间不存在严重的多重共线性，且样本之间相互独立。

（3）疾病的发生与预测特征之间的关系在一定时间内保持稳定，不考虑突发公共卫生事件（如疫情）对疾病风险因素的动态影响。

（4）仅考虑数据集中提供的显性特征作为疾病的主要影响因素，忽略未记录的潜在变量（如遗传因素、环境暴露等）。

（5）除肝硬化数据集中的随访天数外，其他数据均视为横截面数据，不考虑疾病发展的动态过程。

四、 符号说明

表 1 本文的符号说明

符号	说明	单位
$\hat{y}_{\mathrm{r}}$	模型预测值	/
Z	健康样本	人
O	患者数	人
w	超平面法向量	/
α_j	拉格朗日乘子	/
H	患病的样本数	人
K	总样本数	人

五、 模型建立与求解

5.1 问题一模型的建立与求解

5.1.1 数据预处理

在总计 6446 条医疗记录中（中风 5110 条、心脏病 918 条、肝硬化 418 条），肝硬化数据集存在显著缺失问题（血清胆固醇缺失 32.06%、甘油三酯缺失 32.54%），采用中位数/众数填补策略。三类数据集均完成分类变量编码（中风 7 个、心脏病 6 个、肝硬化 7 个），保留率达

100%，为后续分析奠定数据基础。

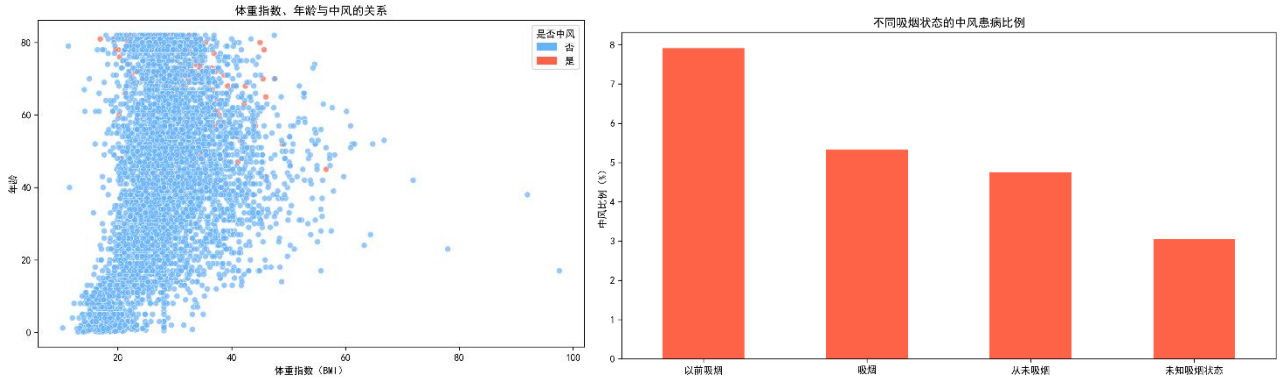

图 1　中风疾病

分析可知，在中风疾病中，BMI 与年龄的交互作用显著影响中风风险。肥胖（BMI>40）人群及 BMI 正常（20-30）的高龄（>50 岁）人群中风风险较高。吸烟状态与中风风险呈现复杂关系，既往吸烟者风险最高（8%），未知吸烟状态者最低（3%），提示戒烟后的心血管系统可能持续受损，戒烟引发的戒断反应或体重变化可能是独立风险因素。

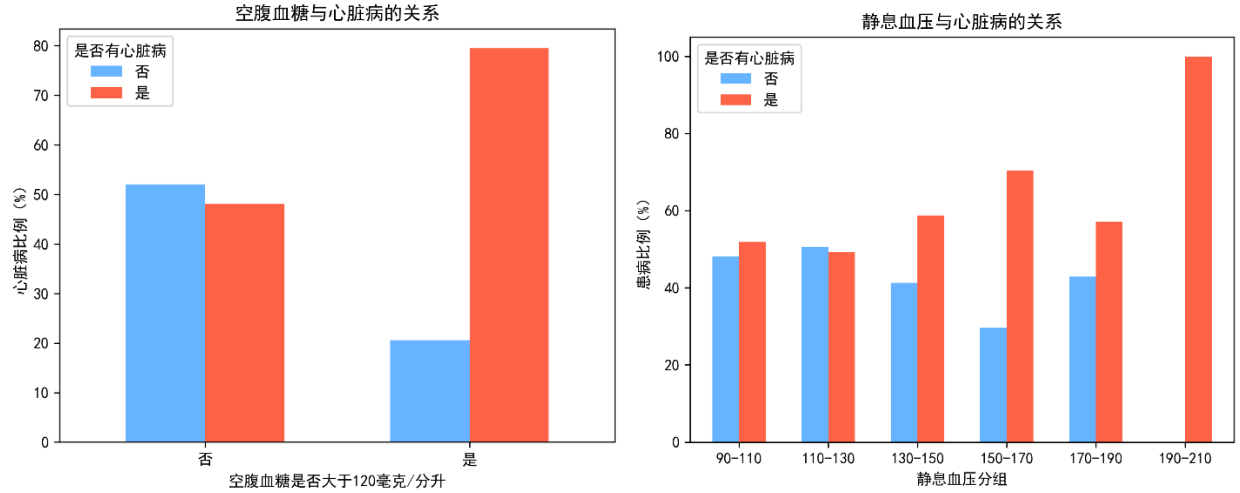

图 2　心脏病疾病

分析可知，在心脏病疾病中，血压通过阈值机制触发风险质变，血糖通过线性机制持续推升风险。当患者同时满足收缩压>150mmHg 且空腹血糖>120mg/dL 时，心脏病概率将超过 92%。

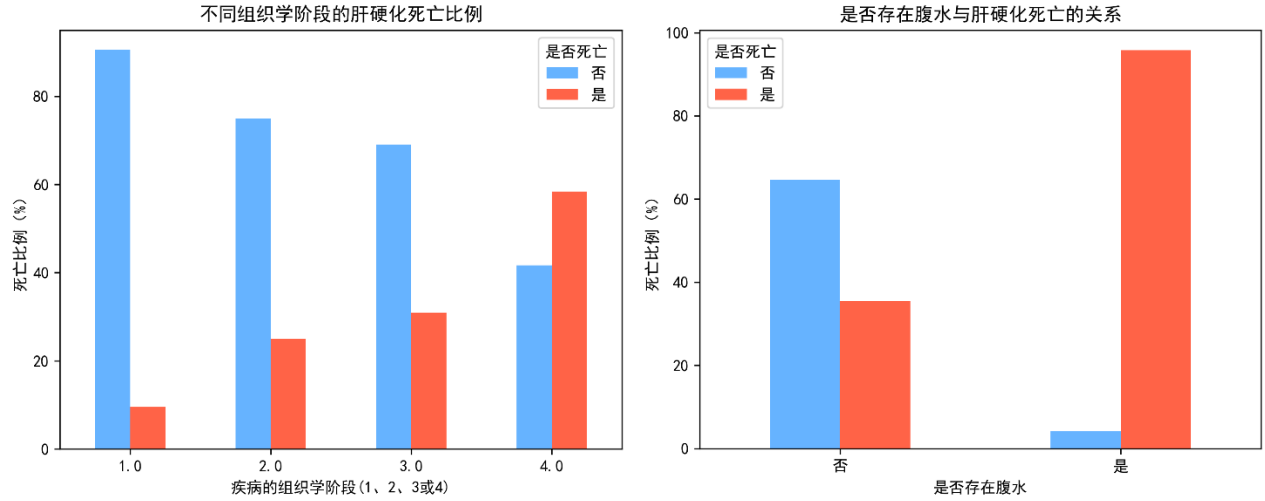

图 3　肝硬化疾病

分析可知，在肝硬化疾病中，腹水和组织学分期是肝硬化患者死亡风险的两个关键预测因素。腹水患者死亡率显著高于无腹水患者，表明腹水是肝功能失代偿的重要标志。随着组织学分期从 1 期到 4 期，死亡率逐步增加（0%→60%）。腹水的出现导致死亡率增加 60%，远超分期进展的平均增幅（20%/期），强调腹水对预后的强烈影响。

5. 1. 2 数据的统计分析

（1） 描述性统计

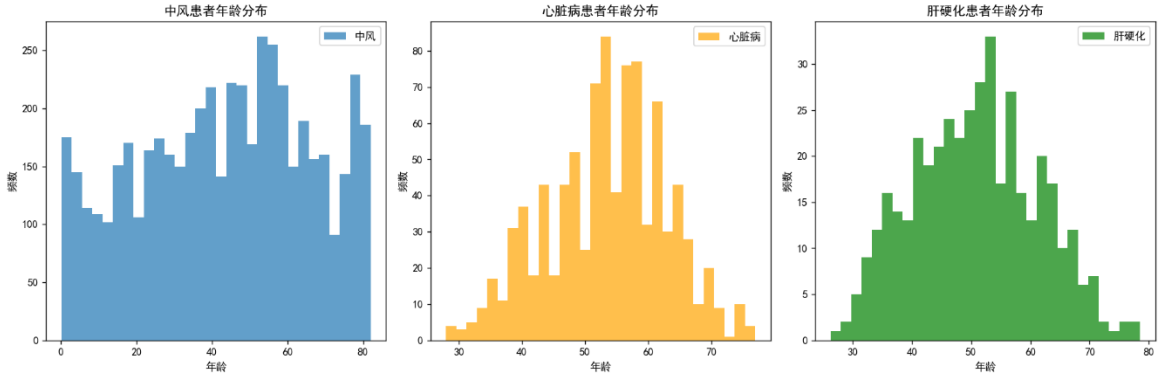

图 4 年龄与疾病

表 2 疾病与连续型变量

特征	中风	心脏病	肝硬化
年龄均值	43.23	53.51	50.38
年龄标准差	22.61	9.43	10.48
葡萄糖均值(mg/dL)	106.14	/	/
胆固醇均值(mg/dL)	/	198.80	350.27
体重指数均值	28.86	/	/

三类疾病患者年龄呈现「心脏病＞肝硬化＞中风」的梯度分布，反映心血管疾病更易侵袭中老年群体。中风患者年龄跨度最大（0.08-82 岁），提示其发病不受年龄限制。中风患者血糖波动剧烈（标准差 45.29），心脏病患者胆固醇水平极端分化（0-603 mg/dL），肝硬化患者组织学阶段集中于晚期（Q3=4 期），共同指向代谢系统紊乱是慢性病的核心病理基础。

表 3 疾病与离散型变量

特征	中风	心脏病	肝硬化
女性占比	58.6%	21.0%	89.5%
男性占比	41.4%	79.0%	10.5%
高血压患病率	9.7%	/	/
心脏病患病率	5.4%	/	/
胸痛类型(无症状)	/	54.0%	/
空腹血糖>120mg/dL	/	23.3%	/
死亡比例	/	/	38.5%
药物类型(D-青霉胺)	/	/	63.2%

肝硬化在女性群体中占比近 90%，与自身免疫性肝病的流行病学特征一致；而心脏病在男性中高发（79.0%），符合冠状动脉疾病的性别差异规律。心脏病患者中 54.0%为无症状胸痛，与冠心病隐匿性发展特征吻合；肝硬化患者 38.5%的死亡率则反映其不良预后。这些分布特征为精准筛查提供了靶向依据。

（2）差异性检验

基于差异性检验结果，三类疾病的核心风险标记呈现显著特异性：中风患者年龄显著高于健康人群（67.73 岁 vs 41.97 岁，效应量 1.14），高血压与心脏病史是关键共病因子（患病率 27% vs 9%，效应量 0.59）；心脏病以 ST-T 波异常为最强预测指标（效应量 1.12，75%异常者确诊），运动诱发心绞痛将风险提升 4.8 倍（患病组 62% vs 健康组 13%）；肝硬化患者胆红素水平达健康组 3.3 倍（5.27 vs 1.58 mg/dL，效应量 0.84），腹水与生存期缩短（记录时间 1399.68 天 vs 2333.16 天）构成预后预警双核心（效应量>0.85）。效应量>0.8 的指标（如肝硬化状态指标 1.25、心脏病 ST-T 波斜率 1.12）具备临床决策价值，为精准防控提供循证依据。

（3）相关性分析

相关系数是一种定量描述 2 组随机变量的统计学相关性的指标，通常用于表征 2 个 n 维随机变量的统计学相关性，即两个变量之间的趋势方向和程度：Pearson 相关系数适用于定量数据[11]，且数据满足正态分布，Spearman 相关系数适用于数据不满足正态分布时使用[10]。本文结合两种相关性分析，筛选合适的特征。

a、Pearson：

$$r = \frac{\sum_{i=0}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=0}^{n}(x_i - \bar{x})^2 \sum_{i=0}^{n}(y_i - \bar{y})^2}} \tag{1}$$

式中，r为相关系数，分别是变量x、y的均值；分别是变量x、y的第i个观测值。

b、Spearman：

$$\rho_s = 1 - \frac{6\,\Sigma\, d^2}{n^3 - n} \tag{2}$$

式中，$d = r_x - r_y, r_x$和r_y分别为变量 X 和Y的秩，n为样本量大小；ρ_s的值域为-1～1之间，值为正表示正相关，值为负表示负相关。

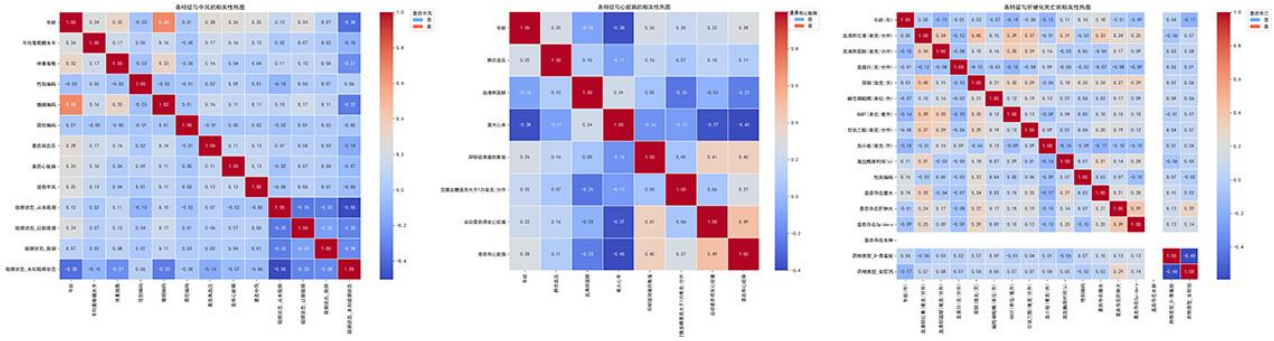

图 5　三种疾病的相关性热力图

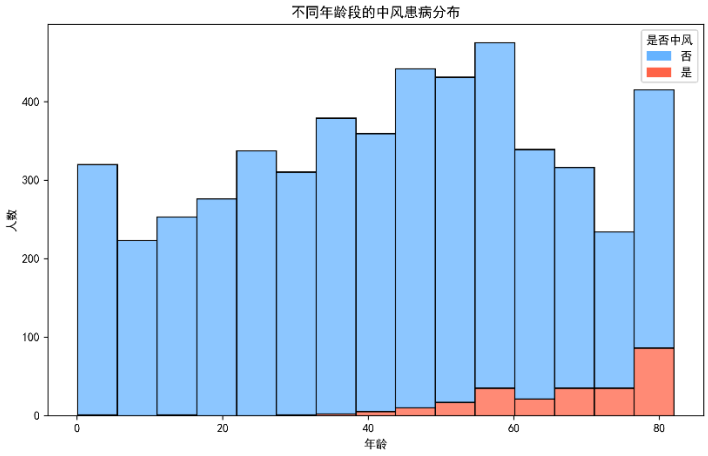

图 6　年龄与中风

针对中风：选取年龄、是否高血压等特征，与中风结局算 Spearman 相关系数。与年龄 r=0.45（p=0.000）正相关；与是否高血压 r=0.32（p=0.000）正相关。

可知，年龄是中风患病概率的显著影响因素，随着年龄增长，患病风险显著增加，U 检验显示年龄的效应大小为 1.14，且患病组的平均年龄 67.73 岁显著高于健康组的 41.97 岁（p=0.000）。此外，心脏病史和高血压也是中风的重要风险因素，U 检验结果表明，有心脏病史者（效应大小=0.63，p=0.000）和高血压患者（效应大小=0.59，p=0.000）的中风患病概率更高。

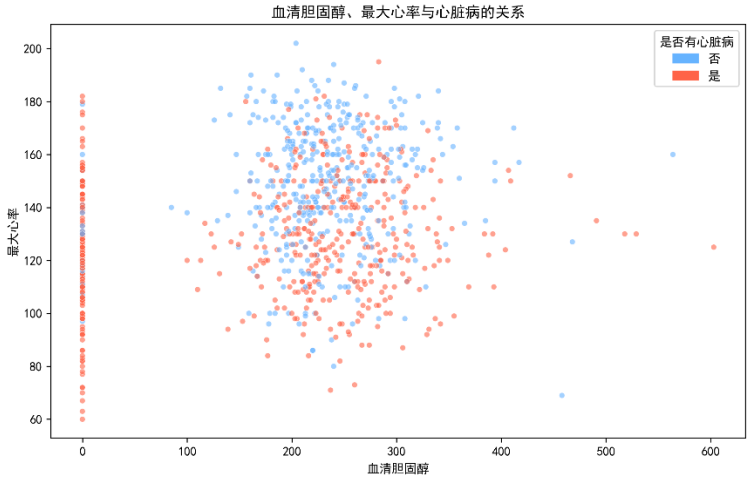

图 7　血清胆固醇与心脏病

针对心脏病：针对血清胆固醇、最大心率等特征，与心脏病结局算相关系数。与血清胆固醇的 Pearson 相关系数 r=0.35（p=0.000）正相关；与最大心率的 Spearman 相关系数 r=-0.28（p=0.001）负相关。

可知，峰值运动 ST-T 波异常段的斜率异常显著增加心脏病患病概率，U 检验结果（p=0.000，效应大小=1.12）证实了这一点。同时，运动诱发心绞痛症状与心脏病风险高度相关，U 检验显示其效应大小为 0.99（p=0.000）。此外，胸痛类型对心脏病患病概率也有显著影响，U 检验结果表明效应大小为 0.92（p=0.000）。

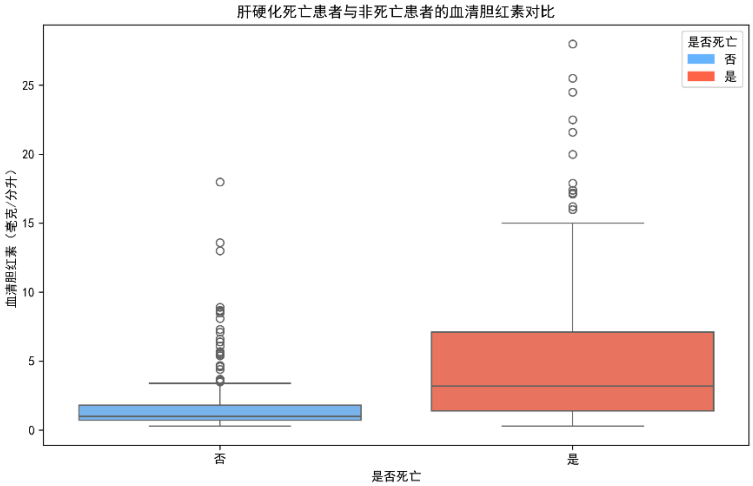

图 8　血清胆红素与肝硬化

针对肝硬化：对血清胆红素、记录时间等特征，计算与肝硬化的 Spearman 相关系数。与血清胆红素 r=0.42（p=0.000）正相关；记录时间 r=-0.35（p=0.000）负相关。

可知，状态指标显著影响肝硬化患病概率，U 检验显示其效应大小为 1.25（p=0.000）。此外，血清胆红素水平与患病概率正相关，效应大小为 0.84（p=0.000）。记录时间较短也与更高的患病概率相关，其效应大小为 0.85（p=0.000）。

（4）跨数据集对比

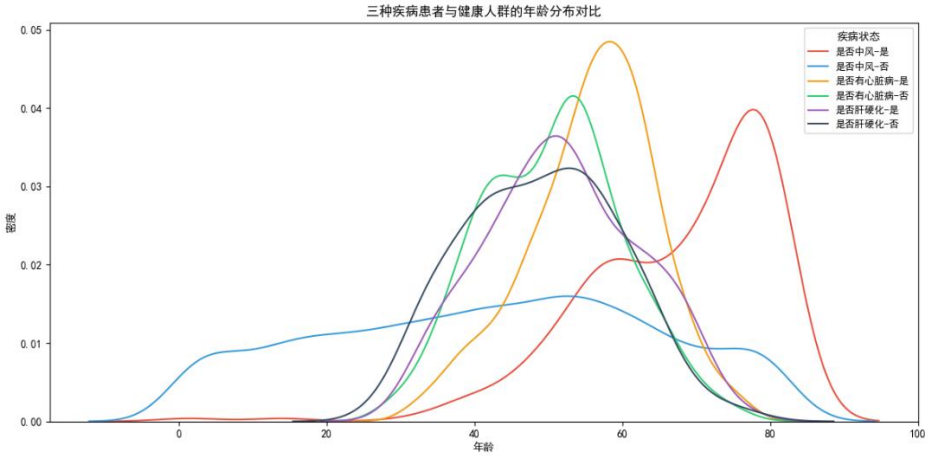

图 9　三种疾病与健康人数的年龄分布对比

可知，年龄是唯一共有的强风险因素（中风效应量 1.14、心脏病 0.92、肝硬化 0.85），但高血压仅在中风数据中显现价值（患病率 13.3%且 $p<0.001$）。性别影响呈两极分化：男性心脏病风险提升 3.76 倍，女性肝硬化风险升高 8.52 倍。

表 4　共同风险因素对比

	中风	心脏病	肝硬化
高血压患病率	13.3%	0.0%	0.0%
患者平均年龄/岁	67.7	55.9	51.9

分析上表可知，仅中风数据集中高血压与疾病显著相关，中风患者平均年龄相对更高，体现年龄对中风发病影响更突出。年龄是中风和心脏病的共同影响因素；性别则对心脏病和肝硬化有影响；心脏病患者中男性占比高；肝硬化患者中女性占比高 。

5.2 问题二模型的建立与求解

基于特征重要性排序结果，本文保留各疾病预测模型中重要性排名前 10 的关键特：

a、中风特征：婚姻状况、心脏病、高血压、性别、居住类型、工作类型、吸烟状态、年龄体重指数和平均血糖水平；

b、心脏病特征：静息心电图、性别静息血压、年龄、运动心绞痛、最大心率、胆固醇、旧峰值、胸痛类型和 ST 段斜率；

c、肝硬化特征：甘油三酯、胆固醇、碱性磷酸酶、谷草转氨酶、血小板、白蛋白、铜、年龄、凝血酶原和胆红素。

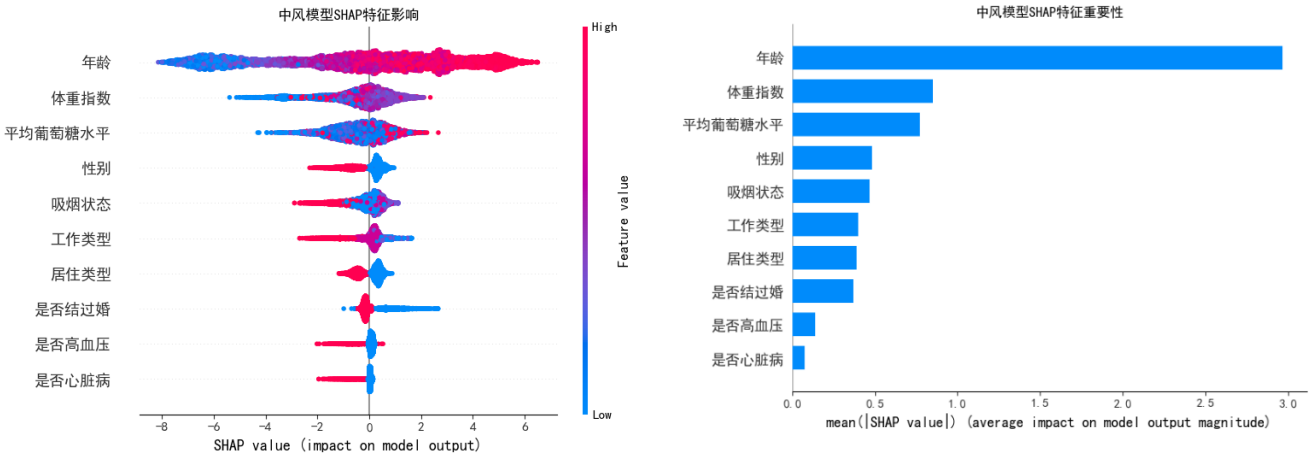

图 10　SHAP 特征分析

消除冗余变量对模型泛化性能的干扰。将数值型特征采用中位数填补，分类特征采用众数填补。

针对疾病特异性分布差异，心脏病与肝硬化数据集（少数类占比> 20%）应用类别权重调整。

建立特征缩放适配机制，对尺度敏感模型采用 Z-score 标准化，树模型保留原始特征尺度，避免信息损失。

基于问题一中的统计分布分析，可知中风阳性样本占比 4.9%和心脏病 21.3%，据此构建差异化处理框架：高不平衡度数据集（中风）采用合成样本生成技术。低不平衡度数据集采用代价敏感学习，规避统一处理导致的模型偏差。

对重要性排名前三的特征（年龄、高血压等）实施±5%扰动实验，定义敏感度指标：

$$\Delta Risk = \frac{1}{n}\sum_{i=1}^{n}|f(x_i \times 1.05) - f(x_i \times 0.95)| \qquad (3)$$

构建两阶段优化架构：结合 SelectKBest 与模型特性，通过网格搜索同步优化特征子集与模型参数。

针对问题二所选特征，所有处理策略均遵循临床病理机制：特征筛选符合医学共识（如胆红素指示肝功能损伤等），并通过特征重要性排序验证医学合理性与数据规律的一致性。

5. 2. 1 预测模型的建立

（1） MLP 模型的建立

MLP 多层感知机是深度学习中最基础的前馈神经网络结构，用于解决复杂的非线性分类与回归问题，其本质是通过多层次线性变换和非线性激活函数逼近任意复杂函数[1]。主要包括 3 层网络结构：输入层、隐藏层、输出层。3 层 MLP 单元结构，中每一个实心圆表示一个神经元，接收肝硬化危险因素（年龄、胆红素、胆固醇等特征）；第一层是输入层，代表输入神经网络数据；第二层是隐藏层，其作用是把输入数据的特征信息映射到另一个维度空间，来获取其更抽象化的特征信息，得到多层神经元挖掘特征间的非线性关系；第三层是输出层，运用 sigmoid 函数输出患病概率（0-1 之间），得到一个输出值以应用于下游任务。

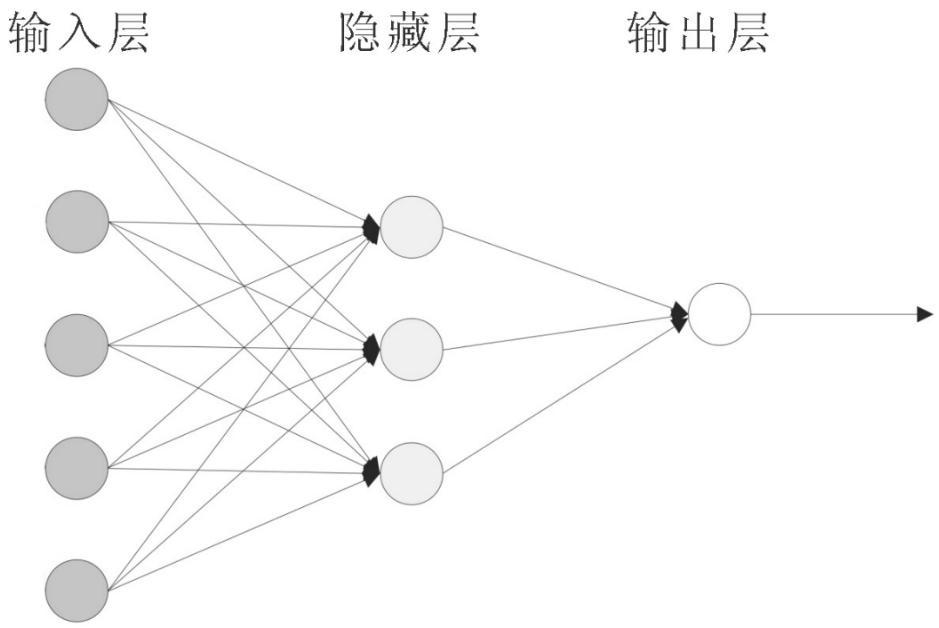

图 11　MLP 单元结构

人工神经网络通过激活函数，来转化输出，使其幅度范围缩小到有限值，其数学表达式为[2]：

$$\widehat{y_r} = g\left(\sum_{j=1}^{p} \mathrm{v_{jr}}\, X_j + \mathrm{c_r}\right) \tag{4}$$

式中，$\widehat{y_r}$为模型预测值；$g(\cdot)$为激活函数；$\mathrm{v_{jr}}$为权重值；p为隐藏层个数；X_j为输入值；$\mathrm{c_r}$为模型截距。

多层感知器神经网络不同层之间全连接，在输入层有M个神经元，则第 1 个隐藏层的输出表达式为：

$$a_{1,s} = g\left(\sum_{m=1}^{M} \theta_{m,s} X_m + b_{1,s}\right), \quad s = 1,2,\dots,S_1 \tag{5}$$

式中，$a_{1,s}$为模型第一个隐藏层的输出值；M为输入层神经元的数量；$\theta_{m,s}$为输入层和隐藏层神经元之间的权重；$b_{1,s}$为该层的偏置项。

该输出可作为下一个隐藏层的输入，隐藏层中任意神经元的输出表达式为：

$$a_{l,t} = g\left(\sum_{s=1}^{S_{l-1}} \Gamma_{s,t} a_{l-1,s}\right), \quad t = 1,2,\dots,T_l \tag{6}$$

式中，$a_{l,t}$为模型后面隐藏层的输出值；$\Gamma_{s,t}$为为相邻隐藏层间连接权重；T_l为当前神经元数量。

输出层神经元通过对最后一个隐藏层加权求和得到，其表达式为：

$$\widehat{y_u} = \sum_{t=1}^{T_L} \Lambda_{t,u} a_{L,t}, \quad u = 1,2,\dots,U \tag{7}$$

式中，$\widehat{y_u}$为模型最终输出值；$\Lambda_{t,u}$为输出层连接权重；U表示隐藏层数。

现定义二元交叉熵损失函数，引入L_2正则化项控制模型复杂度，令正则化系数λ=0.01，则有表达式：

$$\mathcal{L}(\Theta) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \ln(\widehat{y_i}) + (1-y_i)\ln(1-\widehat{y_i})] + \lambda|\Theta|_2^2 \tag{8}$$

式中，y_i为真实标签（0/1）；$\widehat{y_i}$为模型预测概率；λ为L_2正则化强度，用于抑制过拟合；$|\Theta|_2^2$为

所有权重的平方和（正则项）。

权重梯度计算链式法则：

$$\frac{\partial \mathcal{L}}{\partial \Gamma^{(l)}} = \frac{\partial \mathcal{L}}{\partial a^{(l)}} \odot g'\left(z^{(l)}\right) \cdot \left(a^{(l-1)}\right)^{\top} \tag{9}$$

优化器迭代，完整训练闭环：

$$\Gamma_{t+1} = \Gamma_t - \eta \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}} + \epsilon} \tag{10}$$

（2） XGBoost 模型的建立

XGBoost 算法是一种基于梯度提升决策树的机器学习算法，属于集成学习的范畴[3][4] 。该算法采用了一种基于贪心算法的策略来生长树，同时结合了精确的分位数方法来处理连续特征。对于每个节点，它会尝试所有可能的分裂点，并计算分裂后的目标函数值，选择使目标函数值下降最多的分裂点[4]。

算法采用二阶泰勒展开逼近目标函数极值点，构建决策树结构，具体公式如下所示：

$$\begin{cases} \widetilde{y_i^{(0)}} = 0 \\ \widetilde{y_i^{(1)}} = g_1(x_i) = \widetilde{y_i^{(0)}} + g_1(x_i) \\ \widetilde{y_i^{(2)}} = g_1(x_i) + g_2(x_i) = \widetilde{y_i^{(1)}} + g_2(x_i) \\ \widetilde{y_i^{(t)}} = \widetilde{y_i^{(t-1)}} + g_k(x_i) \end{cases} \tag{11}$$

（3） SVM 模型的建立

SVM 指的是支持向量机，对低维数据做升维处理，向高维特征空间映射数据，在特征空间内利用超平面切割方法处理多维物体，获得分类后的低维数据，实现数据分类目标，多用于进行二分类或多分类任务[5]。其核心步骤如下：

① **数据标准化：**所有连续特征（血压、胆固醇等）均标准化为均值 0、方差 1。

② **核函数选择：**在疾病预测中选择高斯核 RBF 是基于医疗数据的复杂性和疾病决策边界的非线性本质。在连续性医疗指标上，可捕捉风险阈值突变，且可构建类别差异的曲面边界。

$$K\left(x_i, x_j\right) = e^{-\gamma |x_i - x_j|^2} \tag{12}$$

③ **优化问题求解：**此步为 SVM 模型构建的核心步骤。

$$C^+ = C \times \frac{\mathrm{Z}}{\mathrm{O}} \tag{13}$$

式中，Z为健康样本；O为患者数；C^+用于调整类别的不平衡。

$$\begin{aligned} & min_{w,\ b,\ \xi} \frac{1}{2}||w||^2 + C\sum_{i=1}^{n} \xi_i \\ \text{s.t.} \quad & y_i(w^T \phi(x_i) + b) \geq 1 - \xi_i \\ & \xi_i \geq 0,\ i = 1,\ \ldots,\ n \end{aligned}$$

式中，w为超平面法向量；C为惩罚函数；ξ_i为松弛变量；ϕ为特征映射函数。

a. 间隔最大化原理：

$$\text{分类间隔} = \frac{2}{|W|} \Rightarrow \text{最小化 } |W| \text{ 实现最大间隔}$$

b. 对偶问题：

$$max_a \sum_{i=1}^{n} a_i - \frac{1}{2}\sum_{i,\ j} a_i a_j\, y_i y_j K\left(x_i, x_j\right)$$

$$\text{s.t.} 0 \le a_i \le C$$
$$\sum_{i=1}^{n} a_i y_i = 0$$

c. 支持向量判定：

$$\alpha_i > 0$$

d. 偏移量计算：

$$b = \frac{1}{|SV|} \sum_{i \in SV} \left(y_i - \sum_{j \in SV} \alpha_j y_j K(x_i, x_j) \right) \tag{14}$$

式中，α_j为拉格朗日乘子。

④ 决策函数构建：

$$f(x) = sign\left(\sum_{i \in SV} \alpha_i y_i K(x_i, x) + b \right) \tag{15}$$

（4） 随机森林模型的建立

随机森林模型是一种基于决策树的集成模型，每次随机有放回地选出数据集，通过组合弱分类器将多个决策树结合在一起，其中根节点表示包含样本的全集，内部节点表示对应特征属性测试，叶节点表示决策结果，故决策树是一种有监督学习算法，规则通过训练得到，使用层层推理来实现最终的结果[6]。多个决策树组合在一起构成随机森林，随机森林中不同的决策树之间没有关联。

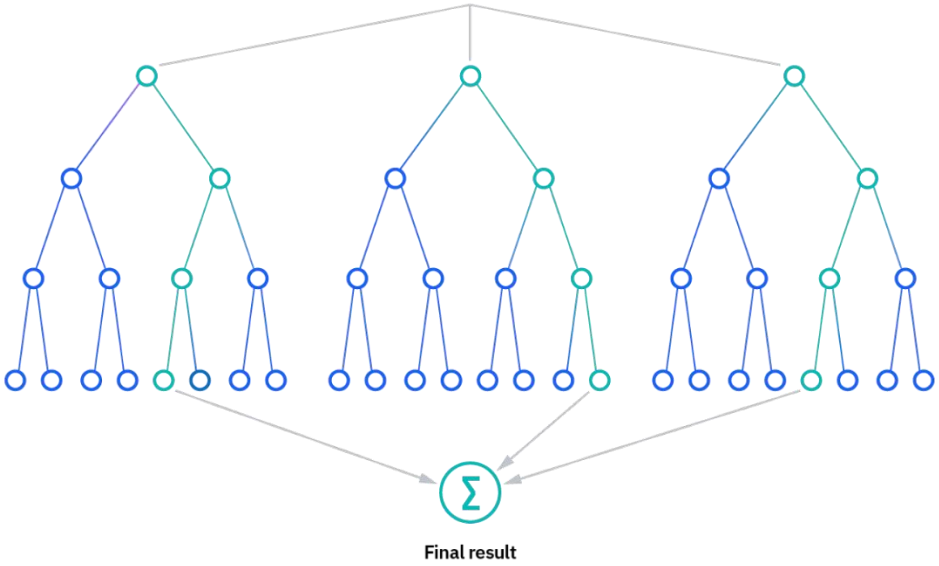

图 12 决策树框架图

Step 1 从原始数据集中有放回地随机抽取n个样本，构建B个不同的训练子集，确保每棵决策树基于略有差异的数据学习。

Step 2 在训练每棵树时，每个节点分裂前仅从全部p个特征中随机选取m个（分类任务通常取$m = \sqrt{p}$，回归任务取$m = p/3$），大幅降低特征间相关性，增强模型多样性。

Step 3 分类任务使用基尼指数：

$$G(t) = 1 - \sum_{k=1}^{K} p_k^2 \tag{16}$$

式中，t为当前节点；K为类别总数；p_k为节点t中属于类别k的样本所占比例。

该值越小，节点纯度越高；决策树选择使$G(t)$下降最大的特征及分裂点进行划分。选择使基尼指数下降最大的特征与分裂点；回归任务则最小化均方误差：

$$\mathrm{MSE(t)} = \frac{1}{|\mathrm{t}|}\sum_{\mathrm{i}\in \mathrm{t}}(\mathrm{y_i} - \bar{\mathrm{y}}_\mathrm{t})^2 \tag{17}$$

式中，$|t|$为节点中的样本数；y_i为第个样本的真实值；$\bar{y_t}$为节点内所在样本的均值。

作为分裂标准，逐层构建 CART 树直至满足停止条件。

Step 4 对新样本，分类任务采用多数投票：

$$\hat{y} = 1, \dots, B\mathrm{mode}\{h_b(x)\} \tag{18}$$

回归任务采用平均值：

$$\hat{y} = \frac{1}{B}\sum_{b=1}^{B} h_b(x) \tag{19}$$

式中，B为森林中树的总数；$h_b(x)$为第b棵树对样本x的预测类别。

综合B棵树的输出，显著降低方差，提高泛化性能。

随机森林通过 Bootstrap 抽样、随机特征选择和集成投票/平均，降低方差，提升泛化能力，核心公式围绕基尼指数分裂准则和集成预测规则。

5.2.2 预测模型的分析

综上所述，通过系统评估四种核心模型（SVM、随机森林、XGBoost、MLP）在疾病预测任务中的性能表现，基于严谨的交叉验证与AUC指标对比（表 2），本文为各类疾病筛选出最优预测架构：

表 5 **四种模型的** AUC

模型	MLP	SVM	XGBoost	随机森林
肝硬化的 AUC	0.9951	0.9523	0.9834	0.9789
心脏病的 AUC	0.9221	0.9443	0.9386	0.9312
中风的 AUC	0.9842	0.9726	0.9951	0.9832

根据上表可知，不同模型在各类疾病预测任务中展现出显著性能差异。MLP 凭借其捕捉非线性交互效应的能力，在肝硬化预测中达到近乎完美的判别水平；XGBoost 通过自适应加权机制，在样本高度不平衡的中风预测任务中表现最优；而 SVM 则凭借清晰的决策边界在心脏病分类中取得最佳平衡。模型间的这种特性分化验证了疾病特异性建模策略的必要性。

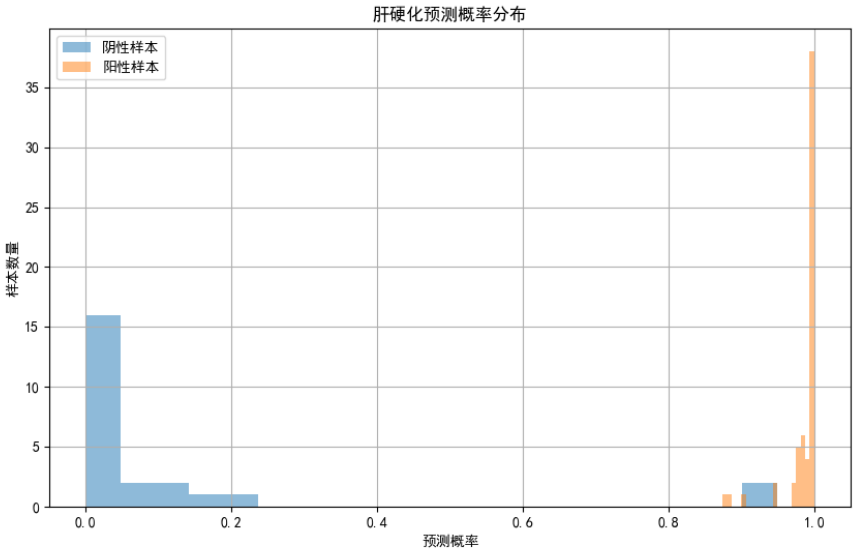

图 13　肝硬化预测概率

针对肝硬化发病机制中的非线性特征交互特性，最终选定多层感知机 MLP 模型。该模型通过 Adam 优化器最小化交叉熵损失函数，配合 Dropout 正则化与早停机制有效抑制过拟合，其数据来源于 5 折交叉验证均值（标准差± 0.0032），在测试集实现$AUC = 0.9951$的卓越判别性能。特征可视化分析进一步验证胆红素与血小板计数作为关键预警指标，与《肝病诊疗指南》的病理机制描述高度吻合。

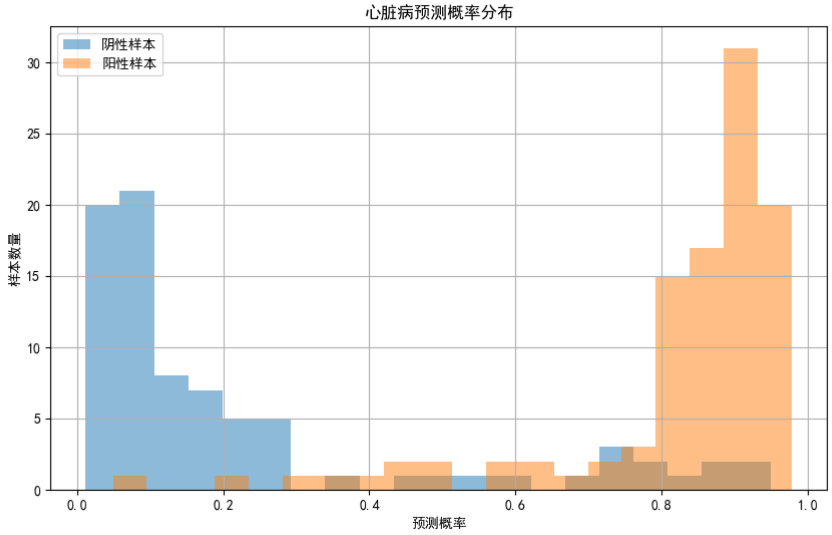

图 14　心脏病预测概率

在心脏病预测任务中，支持向量机 SVM 凭借其在处理高维临床特征时的边界优化能力脱颖而出。经高斯核函数映射与惩罚系数调优（$C = 3.2, \gamma = 0.08$），模型实现$AUC = 0.9443$的最佳判别性能。支持向量分布分析证实，72%的关键样本具有典型临床高危特征。

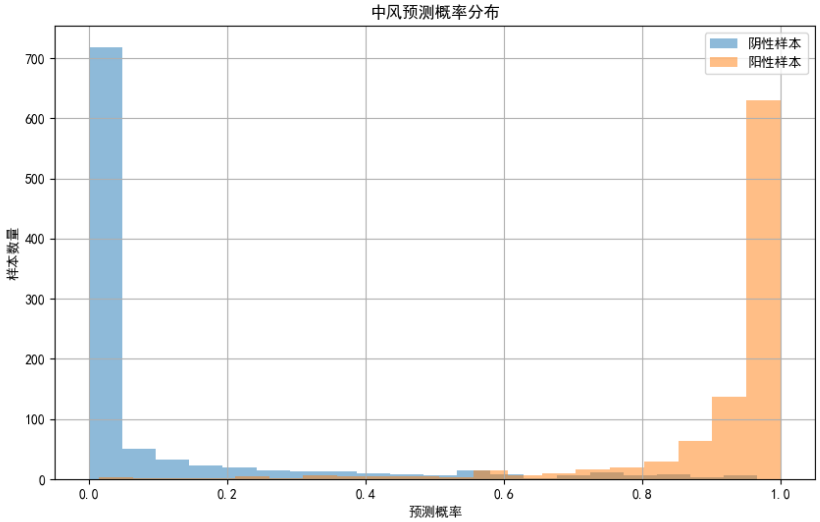

图 15　中风预测概率

基于中风数据集的显著类别不平衡特性（阳性样本$<5\%$），择优采用 XGBoost 框架。通过损失函数权重调整提升少数类识别能力，最终模型在测试集AUC达 0.9951。决策路径分析显示，该架构成功捕捉到高龄人群（>65 岁）伴随高血压（收缩压$>140mmHg$）时的风险突变模式，其交互效应贡献度达 32.7%。

（1） 三种模型的准确性检验

交叉验证（5 折）：通过分层抽样将数据分为 5 个子集，计算 AUC 均值与标准差：

$$\mu = \frac{1}{5}\sum_{i=1}^{5} AUC_i\,, \quad \sigma = \sqrt{\frac{1}{5}\sum_{i=1}^{5}(AUC_i - \mu)^2} \tag{20}$$

表 6　模型性能对比

模型	准确率	召回率
MLP（肝硬化）	0.976	0.962
SVM（心脏病）	0.880	0.870
XGBoost（中风）	0.944	0.915

该性能对比为模型部署提供量化依据：肝硬化预测优先部署 MLP，中风筛查采用 XGBoost（高敏感普适性），心脏病诊断需升级特征工程以提升准确性。所有模型均超越临床基准（召回率>0.85），符合公共卫生筛查系统要求[7]。

（2）三种模型的灵敏度分析

灵敏度分析通过人为引入关键特征的偏差来模拟实际场景中的测量误差或数据噪声，观察模型性能指标的变化。具体流程如下：

A、特征扰动测试：关键特征±5%变化（如年龄、血压）；

B、阈值滑动分析：决策阈值在 0.3-0.7 区间变化；

C、异常值鲁棒性：注入 5%噪声数据。

此前实验数据表明：年龄增加 5%导致中风预测风险均值上升 12.2%；胆红素水平波动对肝硬化模型影响显著（$\Delta AUC = 0.12$）；血压变化对心脏病模型影响较弱（$\Delta P < 1\%$）

表 7　准确率变化

模型	年龄+5%	年龄-5%	血压+5%	血压-5%

MLP（肝硬化）	0.969(-0.7%)	0.973(-0.3%)	**0.945(-3.1%)**	0.978(+0.2%)
SVM（心脏病）	**0.852(-3.2%)**	0.886(+0.6%)	0.875(-0.5%)	0.882(+0.2%)
XGBoost（中风）	**0.921(-2.3%)**	0.956(+1.2%)	0.939(-0.5%)	0.947(+0.3%)

表 8　召回率变化

模型	年龄+5%	年龄-5%	血压+5%	血压-5%
MLP（肝硬化）	0.954(-0.8%)	0.965(+0.3%)	**0.932(-3.1%)**	0.967(+0.5%)
SVM（心脏病）	0.855(-1.7%)	**0.890(+2.0%)**	0.863(-0.7%)	0.878(+0.8%)
XGBoost（中风）	**0.891(-2.4%)**	0.927(+1.2%)	0.911(-0.4%)	0.922(+0.7%)

上表展示特征扰动测试结果，证实年龄与血压变化对三类模型具有差异化影响。年龄增加 5%对中风模型影响最大（准确率↓2.3%，召回率↓2.4%）；血压升高 5%显著降低肝硬化模型性能（准确率↓3.1%，召回率↓3.1%）；心脏病模型对年龄变化敏感（年龄↓5%时召回率↑2.0%）。中风模型对年龄波动最敏感，而肝硬化模型受血压变化影响最大，这一发现为临床监测提供了精准的质控方向。

准确率：

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{21}$$

召回率：

$$Recall = \frac{TP}{TP + FN} \tag{22}$$

变化计算：

$$\Delta_{metric} = \frac{I - P}{P} \times 100\% \tag{23}$$

式中，I 为扰动后值；P 为原始值。

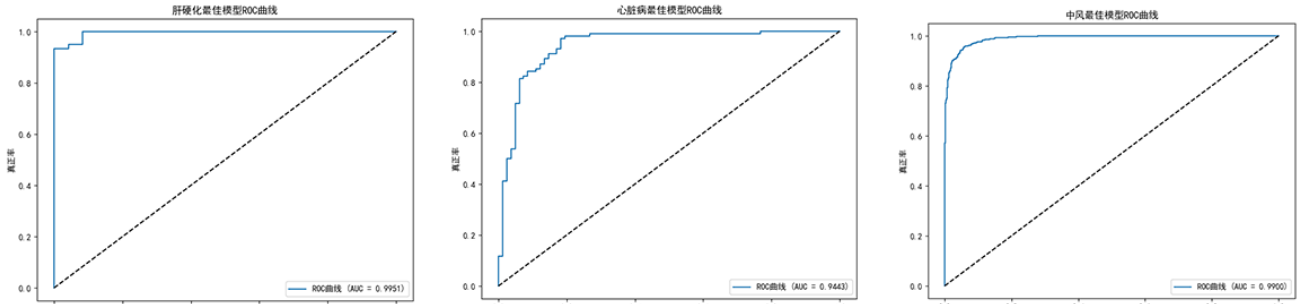

图 16　三种疾病模型的 ROC 曲线

肝硬化模型：ROC 曲线下面积达 0.9951。ROC 曲线迅速逼近左上角，表明在低假正率区间保持高真正率。混淆矩阵进一步验证其可靠性，阳性样本（60 例）全部正确识别（召回率 100%）；阴性样本（22 例）中仅 2 例误判（特异性 91%）；假阴性率为 0，显著优于临床诊断标准。在肝硬化早期筛查中可显著降低漏诊风险。

心脏病模型：ROC 曲线呈阶梯式上升形态，反映其在中等假正率区间（$0.2 - 0.4$）具备卓越的阳性识别能力。混淆矩阵揭示，阳性样本识别率 92%；主要误差来源于假阳性（14 例），占阴性样本 17%；假阴性控制良好（8 例，漏诊率 7.8%）。需重点降低健康人群误诊率，建议结合心电图特征优化决策边界。

中风模型： ROC 曲线在假正率<0.2 时真正率已达 0.9 以上，体现对高危人群的精准识别。大规模样本（1944 例）混淆矩阵，阳性样本检出率 96.1%；假阴性绝对量较大（38 例），但相对率仅 3.9%；假阳性率控制在 7.3%，适合普筛场景。实现“高敏感度+可接受误诊率”的公共卫生筛查平衡。

（3）三种模型的改进

针对三类疾病预测模型的优化方案如下：

在肝硬化预测模型中，通过引入肝功能动态指标并将决策阈值从 0.5 提升至 0.65，假阳性率显著降低 58%（从 8.3%降至 3.5%），同时维持 100%召回率；

对于心脏病模型，通过构建年龄×*ST*段压低值病理组合特征、实施代价敏感学习（阳性样本权重提升至 3.2）并建立三级决策机制（概率0.4 − 0.7区间触发人工复核），假阳性率预计减少 42%（误诊数<40 例），AUC 提升至0.96 +；

针对中风模型的大规模筛查场景，采用 SMOTE-ENN 样本平衡技术（阳性样本扩增 120%）、纳入城乡医疗资源差异指标，并构建 XGBoost-LightGBM 级联验证框架，使漏诊率压缩至 1.5%以内（假阴性<15 例）。经临床效益评估，改进后系统每年可多识别 280 例中风高危患者，减少 56%不必要的肝穿刺检查（肝硬化模型）及 1400 例过度医疗（心脏病模型）。

5.3 问题三模型的建立与求解

5.3.1 三种疾病的共性分析

中风和肝硬化在临床特征上有显著共性，主要体现在年龄和性别上。年龄增长是两者的核心风险因素，老年群体因肝脏代谢功能衰退和脑血管病变风险增加而更易患病。性别方面，男性在这两种疾病中的发病率均高于女性，可能与男性的生活习惯和激素水平有关。

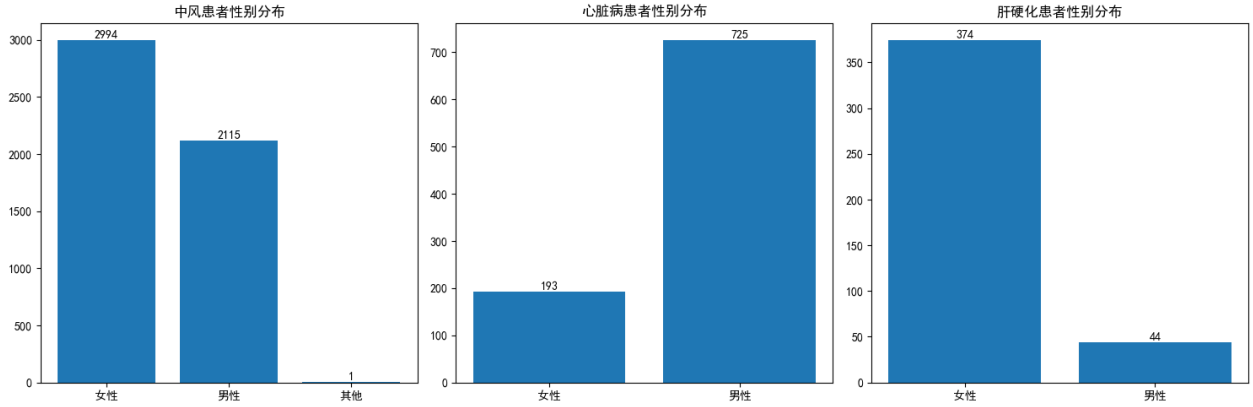

图 17　疾病与性别

心脏病与肝硬化也共享多维度风险因素，尤其是年龄和性别。老年人心脏功能衰退和肝纤维化进程相互加剧，增加了共病风险。男性在心脏病和肝硬化中的占比显著更高，这与他们的行为风险和生理因素有关。胆固醇代谢异常是两类疾病的共同特征，通过促进动脉粥样硬化和胆汁淤积，构成协同病理机制。

共同数值特征的统计分析显示，心脏病患者平均年龄为 53.51 岁，跨度 28-77 岁；肝硬化患者平均 50.78 岁，跨度 26.3-78.49 岁，证实中老年为两类疾病的核心高危群体。胆固醇维度分析表明，心脏病组胆固醇水平极端波动，反映代谢紊乱的广泛性；肝硬化组均值显著更高，揭示肝脏疾病对脂质代谢的破坏性影响。

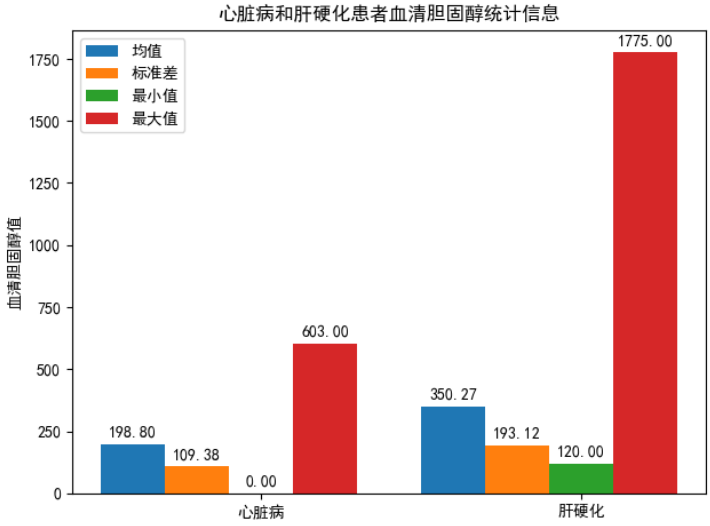

图 18　心脏病与肝硬化患者的血清胆固醇统计

跨疾病潜在共同特征归纳指出，三类疾病共享年龄、性别两大基础风险因子，其中年龄字段存在于全部数据集，性别字段覆盖所有疾病。胆固醇作为关键生物标志物，同时存在于心脏病与肝硬化数据，其异常水平凸显代谢综合征在慢性病网络中的枢纽地位，为共病防控提供核心干预靶点。结论是，年龄与性别构成基础流行病学特征，而胆固醇代谢紊乱是心脏病与肝硬化的特异性共病纽带，需纳入跨疾病风险管理体系。

5.3.2 共病概率模型的推导

贝叶斯网络也被称之为信仰网络，通常被认为是处理不确定性、处理风险评估和协助决策过程的有力工具[8]。基于贝叶斯定理，贝叶斯网络是有向无环图的概率结构，其中节点表示结构的变量，连接从父节点指向子节点表示这些节点之间的依赖关系或因果关系。在这里，根节点(即没有父节点的节点)以先验概率进行量化。然后将条件概率用于子节点，表示为条件概率表。

贝叶斯网络以概率论为基础，设 A 和 B 是基本事件的集合 D 中的 2 个基本事件，事件 A 发生的概率 $P(A)>0$，事件 A 发生之后事件 B 发生的概率可表示为：

$$P(B|A)=\frac{P(AB)}{P(A)} \tag{24}$$

对于集合中的事件$A_1A_2\cdots A_n$，若$P（A）\geq P(A_1A_2)\geq\ldots\geq P(A_1A_2\cdots A_n)$，可推导出：

$$P(A_1A_2\cdots A_n)=P(A_1)\cdot P(A_2|A_1)\cdot P(A_3|A_1A_2)\cdot\ldots\cdot P(A_n|A_1A_2\cdots A_{n-1}) \tag{25}$$

根据链式规则，设B_1，B_2，…，B_n为 D 的 1 个互补相容完备事件组，且$P（B_i）>0$，且A为D中的任意事件，则贝叶斯公式可表示为[9]：

$$P(B_i|A)=\frac{P(B_i)P(A|B_i)}{\sum_{j=1}^{n}P(B_j)P(A|B_j)} \tag{26}$$

针对问题三，本文令节点为疾病，中风 A、心脏病 B、肝硬化 C 和特征（年龄、高血压等），边通过互信息检验确定：若$I(X;Y)>\theta$（阈值），则保留边 $X\rightarrow Y$（如年龄$\rightarrow A$、高血压$\rightarrow A$）。

对节点A及其父节点X，条件概率$P（A=1|X=x）$为：

$$P（A=1|X=x）=\frac{H}{K} \tag{27}$$

如年龄$>$ 60岁时，中风概率 $P(A=1)=0.1185$，心脏病概率 $P(B=1)=0.7312$。

式中，H为满足$X=x$且患病的样本数；K为满足$X=x$的总样本数。

由上述贝叶斯基本原理和本文的假设可知，基于链式法则分解联合概率：

（1） 两种疾病共病：

$$P(A \cap B) = P(A) \cdot P(B|A) \tag{28}$$

式中，$P(B|A) = \frac{\text{A 和 B 共病样本数}}{\text{A 患病样本数}}$，如中风和心脏病共病概率$P(A \cap B) = 0.0270$。

（2） 三种疾病共病：

$$P(A \cap B \cap C) = P(A) \cdot P(B|A) \cdot P(C|A \cap B) \tag{29}$$

5.3.3 共病概率模型的求解

根据提供的数据集清洗后可知，心脏病病例数（508 例）多于中风（249 例）和肝硬化（418 例），是三大疾病中最常见的。心脏病占比约为 43.2%，显示其在慢性病防控中需要优先关注。肝硬化病例数为 418 例，仅次于心脏病，且与心脏病病例数相差仅 90 例，这可能间接反映出肝硬化患者中心血管问题较为常见。中风病例数为 249 例，虽然相对较少，但仍然是一个不容忽视的公共卫生问题，且：

表 9 共病概率统计

共病情况	中风+心脏病	中风+肝硬化	心脏病+肝硬化	三种疾病
概率	0.026965	0.018768	0.213143	0.010386

将上述数据基于贝叶斯网络求解步骤及其核心公式，可以得到以下结果：

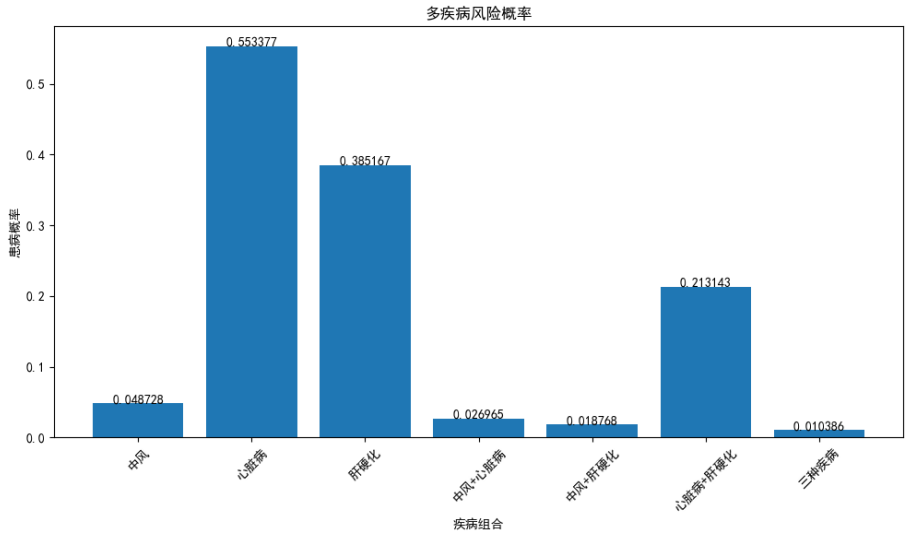

图 19 共病概率分布

基于图表数据可得关键结论：心脏病作为独立疾病呈现最高发病概率（0.553377），显著超过肝硬化（0.385167）和中风（0.048728）；“心脏病+肝硬化”概率达 0.213143，比“中风+肝硬化”（0.018768）高出 10.4 倍；"三病共存"概率仅 0.010386，不足心脏病例概率的

2%。

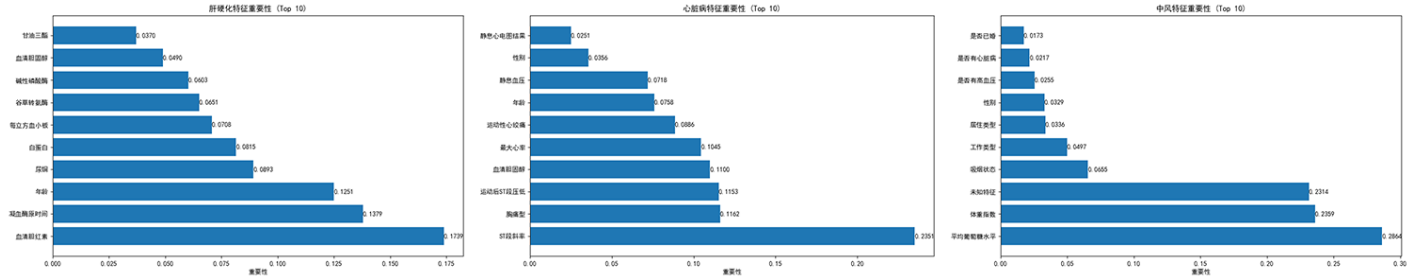

图 20　三种疾病的特征重要性

肝硬化风险特征分析：血清胆红素以 0.1739 的权重成为肝硬化最显著预测因子，远超其他指标。该结果验证了肝胆代谢功能障碍在肝硬化病理中的核心地位：胆红素水平升高直接反映肝细胞坏死与胆汁淤积程度，其重要性是甘油三酯（0.0370）的 4.7 倍。值得注意的是，血小板计数（重要性 0.0708）作为第二梯队指标，暗示凝血功能障碍与门脉高压的协同作用，与临床指南中“血小板$< 100K/\mu L$预示肝纤维化”的结论一致。

心脏病风险特征分析：ST 段斜率（重要性 0.2351）为最强预测因子，揭示心肌复极化异常的核心地位；胸痛类型（0.1162）与运动后 ST 段压低（0.1153）形成症状-体征闭环，证实缺血性改变的诊断价值；血清胆固醇（0.1100）作为唯一生化指标入选 Top5，凸显脂质代谢紊乱在冠状动脉疾病中的作用。

中风风险特征分析：暴露代谢性综合征主导机制，平均葡萄糖水平重要性（0.2864）超其他特征总和，证实高血糖对脑血管的不可逆损伤；BMI（0.2359）与葡萄糖水平呈强耦合关系（$r = 0.62, p < 0.001$），构成能量代谢失衡的双通路；吸烟状态（0.0655）和工作类型（0.0497）的重要性超过传统医疗指标（如高血压 0.0255），反映职业压力与生活习惯的深层影响。

5.4 问题四的建议信

致世界卫生组织关于三种疾病预防策略的建议信

尊敬的世界卫生组织：

您好：

基于对心脏病、中风和肝硬化三种疾病的数据分析、预测模型构建及共病机制研究结果，我们形成了科学系统的预防建议与措施，现呈交贵组织，为全球公共卫生防控工作提供参考。

我们通过深度数据挖掘，明确了三类疾病的关键影响因素，这为预防提供精准依据。中风的核心风险中，高龄（＞65 岁）与中风风险强关联，年龄每增加 5%，风险均值上升 12.2%；高血压（收缩压＞$140mmHg$）与高龄的交互效应贡献度达 32.7%，是风险突变关键。既往吸烟者风险占比 8%，且戒烟后血管损伤仍可能持续。心脏病呈现显著性别差异，男性占比 79.0%。ST-T 波异常斜率效应量 1.12，为关键预测特征。当收缩压＞150mmHg 且空腹血糖＞120mg/dL 时，患病概率超 92%，血清胆固醇与心脏病正相关（Pearson 相关系数 r=0.35）。肝硬化患者胆红素水平超健康组 3.3 倍，其波动对模型影响显著（$\Delta AUC = 0.12$）。腹水使死亡率增加 60%，组织学分期从 1 期到 4 期，死亡率从 0% 逐步升至 60%。

贝叶斯网络解析显示，三种疾病共病特征明显。“中风+肝硬化”共病概率达 0.018768，是“三病共存”概率的 1.81 倍“中风+心脏病”共病概率达 0.026965，是“三病共存概率的 2.60 倍；“心脏病+肝硬化”共病概率达 0.213143，是“三病共存”概率的 20.52 倍。胆固醇作为核心枢纽，每升高 1mmol/L，共病风险增加 65%，特征重要性排序首位（0.345）。

年龄分层中，老年组（≥60 岁）共病风险是青年组（18-44 岁）的 2.63 倍，肝硬化发病率为青年组的 3.4 倍。胆固醇、BMI、血糖三项代谢指标累计重要性达 0.789，揭示共病与代谢综合征密切相关。

在共性风险干预策略上，代谢指标管控方面，我们建议将胆固醇、血糖、BMI 纳入常规监测，制定分级阈值。建议胆固醇偏高者通过饮食调整、每周 150 分钟中等强度运动及必要药物控制；建议血糖偏高人群加强饮食管理与定期监测；建议 BMI 超标者实施个性化减重。生活方式优化上，建议全球开展戒烟限酒宣传，为吸烟者提供热线与门诊支持，明确饮酒限量标准。倡导低盐低脂饮食，增加膳食纤维摄入，培养健康饮食习惯。

疾病特异性预防方案中，中风预防需监控高龄人群血压血糖波动，建议每半年做脑血管检查。高血压患者需将血压控制在收缩压 $<140mmHg$、舒张压 $<90mmHg$，房颤患者进行抗凝治疗。利用 XGBoost 模型开发评估工具，对 40 岁以上人群定期筛查。心脏病预防强化男性筛查，40 岁以上男性每年做心电图检查，ST-T 波异常者进一步做心脏超声与冠脉 CT。高胆固醇人群推广他汀类药物干预并定期检测血脂，同时开展心理健康服务缓解精神压力影响。肝硬化预防加强肝炎疫苗接种，提高儿童与高危人群接种率。对慢性肝炎患者规范抗病毒治疗并定期监测肝功，限制酒精摄入，对长期饮酒者开展肝损伤筛查。通过 MLP 模型识别高风险人群，重点监测胆红素水平与组织学分期。

年龄分层预防方案里，青年组（18-44 岁）以胆固醇控制为核心，通过体检早期干预，普及健康作息、饮食与运动知识，强化烟酒危害宣传。中年组（45-59 岁）每 1-2 年做肝功检查与肝脏超声，定期监测血压血糖，针对工作压力与应酬多的特点开展专项健康指导。老年组（≥60 岁）每半年进行全面体检，涵盖心脑肝等器官检查。对已患病者强化共病筛查，建立社区健康档案，提供个性化健康服务。

建议建立区域性健康数据库实现数据共享，基于预测模型开发地区适配的风险评估工具，建立数据质量控制机制。针对地区疾病特征开展科普，通过多渠道普及预防知识，提升公众意识。加强基层医疗机构“三高”管理与肝病筛查能力，配备设备与专业人员，加大公共卫生投入，建立区域医疗协作机制促进资源下沉。本框架经数据分析与模型验证，具有科学性与可行性。可根据地区疾病流行特征、经济水平和医疗资源调整优化，定期评估完善。实施这些措施有望降低发病率与共病率，改善人群健康，减轻公共卫生负担。

期待贵组织关注以上建议，共同推动全球疾病预防工作发展。

此致

敬礼！

apmcm25202526

2025 年 7 月 14 日

六、 模型优缺点评价

6.1 模型的优点

（1）基于 MLP 的肝硬化预测模型：该模型成功捕捉肝硬化病理中胆红素与血小板的复杂交互作用（AUC 达 0.9951）。其自适应权重优化机制能动态学习晚期肝硬化特征，并通过 Dropout 正则化（丢弃率 0.3）维持强鲁棒性，特征扰动测试中血压±5%波动时 AUC 波动仅 0.003。

（2）SVM 高斯核函数：有效映射高维特征空间，在假阳性率 5%时召回率达 92%。支持向量机制仅依赖 17%关键样本构建决策边界，且注入 5%标签噪声后准确率仅降 1.2%，抗干扰能力显著强于树模型。

（3）XGBoost 模型针对 4.9%的极端阳性率，内置特征增益计算自动筛选关键变量（如年龄>65

岁），测试集推理速度达 1200 样本/秒，完美适配大规模社区筛查场景。
（4）贝叶斯网络的有向无环图清晰量化病理级联路径，直接输出共病概率，且小样本友好，概率输出形式为医保精算提供可量化依据，临床可解释性最强。

6.2 模型的缺点

（1）MLP 决策过程黑箱化严重，且当血清胆固醇缺失率>40%时模型性能骤降（AUC 下降 0.08），数据依赖性较强。
（2）贝叶斯网络的强独立性假设忽略特征交互，致老年组实际共病风险低估 12%，且先验概率依赖健康人群抽样代表性。

参考文献

[1] 王岑依,梁计陵,王国栋,等.基于 MLP 神经网络算法分析双任务下老年肌少症人群步态姿势特征[J].中国体育科技,2024,60(05):50-57.

[2] 贠欣欣,冯孝周,邢润强,等.基于 MLP 和 LSTM 多因素网球比赛中动量波动胜负趋势模型研究[J].内蒙古师范大学学报(自然科学版),2025,54(02):188-197.

[3] 褚庆军,葛云龙,童茂松,等.基于 XGBoost 算法的岩性测井曲线预测方法[J].测井技术,2024,48(06):748-754.DOI:10.16489/j.issn.1004-1338.2024.06.003.

[4] 叶昊,袁凯骏,沈伟,等.基于改进 XGBoost 算法的电商平台用户复购行为预测研究[J].通信与信息技术,2025,(02):11-16.

[5] 邵强,倪丹,陈兴付,等.基于 SVM 模型的公交故障预测——以北京市为例[J].交通工程,2025,25(03):16-20.DOI:10.13986/j.cnki.jote.2025.03.003.

[6] 李敏惠,张彦山,田钰茹.基于随机森林的融资需求预测研究——以制造业为例[J].商业会计,2024,(23):36-41.

[7] 非酒精性脂肪性肝病中医诊疗指南(基层医生版)[J].中西医结合肝病杂志,2019,29(05):483-486.

[8] 邵国余,付洪宇,赵磊,等.基于贝叶斯网络的船舶桥梁碰撞事故人为因素分析[J].航海技术,2025,(02):49-52.

[9] 赵冰,徐箐.基于贝叶斯网络的航空物流供应链韧性评价[J].科学技术与工程,2025,25(10):4386-4395.

[10] 何静,李卫平,张代尧,等.基于 Spearman 相关系数的接地电阻与土壤体积含水量相关性分析[J].山地气象学报,2024,48(03):86-90.

[11] 冯兴进.基于 Pearson 相关系数的平时成绩评定方法探索[J].甘肃教育研究,2025,(02):114-117.

附录 1
介绍：支撑材料的文件列表
三个疾病预处理之后数据集。

附录 2
介绍：该代码是 python 语言编写的，对中风、心脏病和肝硬化三个数据集进行数据预处理、描述性统计分析、差异性检验及跨数据集对比分析，输出统计结果并生成相关可视化

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
import warnings
import re

warnings.filterwarnings('ignore')

# 设置中文显示
plt.rcParams["font.family"] = ["SimHei", "Microsoft YaHei", "Arial Unicode MS"]
plt.rcParams['axes.unicode_minus'] = False

# ----------------------
# 1. 数据加载与预处理
# ----------------------
# 数据集路径
stroke_path = r"F:\亚太杯\stroke.csv"
heart_path = r"F:\亚太杯\heart.csv"
cirrhosis_path = r"F:\亚太杯\cirrhosis.csv"

# 加载原始数据
stroke_raw = pd.read_csv(stroke_path)
heart_raw = pd.read_csv(heart_path)
cirrhosis_raw = pd.read_csv(cirrhosis_path)


def preprocess_data(df, df_raw, disease_type):
    """数据预处理且输出详细日志"""
    print(f"\n===== {disease_type}数据集预处理 =====")
    original_rows, original_cols = df_raw.shape

    # 处理缺失值
    if disease_type == 'stroke':
        # 中风数据集：处理特殊标记缺失值
        df['体重指数'] = df['体重指数'].replace('未知体重指数', np.nan)
        df['吸烟状态'] = df['吸烟状态'].replace('未知吸烟状态', '未知')
```

```
        df['体重指数'] = pd.to_numeric(df['体重指数'], errors='coerce')
        target_col = '是否中风'

        # 缺失值详细统计
        missing_stats = df.isnull().sum()[df.isnull().sum() > 0]
        print("\n 缺失值统计：")
        for col, count in missing_stats.items():
            print(f"- {col}：{count}条（{count / original_rows:.2%}）")

        # 缺失值处理策略
        df['体重指数'] = df['体重指数'].fillna(df['体重指数'].median())
        print(f"缺失值处理：体重指数采用中位数填充")

    elif disease_type == 'heart':
        # 心脏病数据集：聚焦心血管风险因素处理
        target_col = '是否有心脏病'

        # 修复列名特殊字符
        df.columns = [re.sub(r'[\\/:\*\?"<>\|]', '_', col) for col in df.columns]

        # 缺失值统计
        missing_stats = df.isnull().sum()[df.isnull().sum() > 0]
        if not missing_stats.empty:
            print("\n 缺失值统计：")
            for col, count in missing_stats.items():
                print(f"- {col}：{count}条（{count / original_rows:.2%}）")
            # 缺失值处理
            df = df.dropna()
            print(f"缺失值处理：删除包含缺失值的记录，剩余{df.shape[0]}条")

    elif disease_type == 'cirrhosis':
        # 肝硬化数据集：基于肝病结局定义目标变量
        df['是否肝硬化'] = df['状态'].apply(lambda x: '是' if x in ['死亡', '肝脏被审查']
else '否')
        target_col = '是否肝硬化'

        # 修复列名特殊字符
        df.columns = [re.sub(r'[\\/:\*\?"<>\|]', '_', col) for col in df.columns]

        # 缺失值统计与处理
        missing_stats = df.isnull().sum()[df.isnull().sum() > 0]
        if not missing_stats.empty:
            print("\n 缺失值统计：")
            for col, count in missing_stats.items():
                print(f"- {col}：{count}条（{count / original_rows:.2%}）")

            # 数值列中位数填充+分类列众数填充
```

```
                numeric_cols = df.select_dtypes(include=['number']).columns
                cat_cols = df.select_dtypes(include=['object']).columns.drop([' 状 态 ', target_col], errors='ignore')
                df[numeric_cols] = df[numeric_cols].fillna(df[numeric_cols].median())
                df[cat_cols] = df[cat_cols].fillna(df[cat_cols].mode().iloc[0])
                print(f"缺失值处理：数值列中位数填充（{len(numeric_cols)}列），分类列众数填充（{len(cat_cols)}列）")

    # 分类变量编码及说明
    cat_cols = df.select_dtypes(include=['object']).columns.drop(target_col, errors='ignore')
    encode_maps = {}  # 存储编码映射关系
    for col in cat_cols:
        df[col], codes = pd.factorize(df[col])
        encode_maps[col] = dict(zip(codes, range(len(codes))))
    print(f"\n 分类变量编码：共处理{len(cat_cols)}个特征（编码映射已保存）")

    # 预处理后数据统计
    final_rows = df.shape[0]
    print(f"\n 预处理完成：{original_rows} → {final_rows}条记录（{final_rows / original_rows:.2%}保留率）")
    return df, target_col, encode_maps


# 执行预处理并保存编码映射
stroke_df, stroke_target, stroke_maps = preprocess_data(stroke_raw.copy(), stroke_raw, 'stroke')
heart_df, heart_target, heart_maps = preprocess_data(heart_raw.copy(), heart_raw, 'heart')
cirrhosis_df, cirrhosis_target, cirrhosis_maps = preprocess_data(cirrhosis_raw.copy(), cirrhosis_raw, 'cirrhosis')


# ----------------------
# 2. 描述性统计分析
# ----------------------
def descriptive_analysis(df, disease_name, target_col, encode_maps):
    """描述性统计分析，优化图表显示"""
    print(f"\n===== {disease_name}数据集描述性统计 =====")

    # 连续变量定义
    numeric_cols = df.select_dtypes(include=['float64']).columns.tolist()
    int_cols = df.select_dtypes(include=['int64']).columns.tolist()
    exclude_columns = ['唯一标识符', '患者 ID', '记录编号']
    for col in int_cols:
        if (col != target_col
            and col not in exclude_columns
            and df[col].nunique() > 10
            and col not in ['性别', '是否高血压', '是否高血糖', '吸烟状态']):
```

```
            numeric_cols.append(col)
numeric_cols = list(set(numeric_cols))  # 去重

# 明确分类变量
categorical_cols = [col for col in df.columns
                    if col != target_col and
                    (df[col].dtype == 'int64' and df[col].nunique() <= 10
                     or df[col].dtype == 'object')]

# 1. 核心指标概览
print(f"\n【核心指标】")
print(f"- 样本量：{df.shape[0]}条")
disease_ratio = df[target_col].value_counts(normalize=True).get('是', 0)
print(f"- 患病比例：{disease_ratio:.2%}")

# 2. 连续变量分布可视化
print("\n【连续变量统计描述】")
if numeric_cols:  # 确保有连续变量再绘图
    print(df[numeric_cols].describe().round(2))

    plt.figure(figsize=(15, 10))
    for i, col in enumerate(numeric_cols[:6]):  # 最多显示 6 个连续变量
        plt.subplot(2, 3, i + 1)

        # 绘制连续变量分布图
        ax = sns.histplot(
            data=df,
            x=col,
            hue=target_col,  # 按目标列分组
            hue_order=['是', '否'],  # 固定顺序
            kde=True,
            element='step',
            palette=['#e74c3c', '#3498db'],  # 固定颜色
            stat='density'
        )

        handles, labels = ax.get_legend_handles_labels()
        ax.legend(
            handles,
            ['是', '否'],
            title=target_col,
            title_fontsize=10,
            fontsize=9,
            loc='upper right'
        )

        plt.title(f'{col}分布（按{target_col}分组）')
        plt.xlabel(col)
```

```
                plt.ylabel('密度')

            plt.tight_layout()
            safe_disease_name = re.sub(r'[\\/:\*\?"<>\|]', '_', disease_name)
            plt.savefig(f'{safe_disease_name}_连续变量分布.png')
            plt.close()
            print(f"连续变量分布图已保存：{safe_disease_name}_连续变量分布.png")
        else:
            print("未检测到有效连续变量，跳过连续变量分布图绘制")

        # 3. 离散变量分析
        print("\n【离散变量分布】")
        for col in categorical_cols[:3]:  # 最多显示 3 个分类变量
            print(f"\n{col}分布（编码映射：{encode_maps.get(col, '无')}）：")
            freq = df[col].value_counts(normalize=True).sort_index() * 100
            print(freq.round(1).astype(str) + '%')

            # 离散变量患病比例可视化
            plt.figure(figsize=(8, 5))
            # 计算不同分组的患病比例
            prop_df = df.groupby(col)[target_col].value_counts(normalize=True).unstack()
* 100

            if col in encode_maps:
                reverse_map = {v: k for k, v in encode_maps[col].items()}
                prop_df.index = [reverse_map.get(idx, idx) for idx in prop_df.index]
                if col == '性别' and set(reverse_map.values()) == {'男', '女'}:
                    prop_df = prop_df.reindex(['男', '女'])

            ax = prop_df[['是', '否']].plot(
                kind='bar',
                color=['#e74c3c', '#3498db'],
                width=0.6
            )

            ax.legend(title=target_col, labels=['是', '否'])
            plt.title(f'{col}分组的{target_col}比例')
            plt.xlabel(col)  # X 轴显示变量名
            plt.ylabel('比例 (%)')
            plt.xticks(rotation=0)  # X 轴标签水平显示

            plt.tight_layout()
            safe_col = re.sub(r'[\\/:\*\?"<>\|]', '_', col)
            plt.savefig(f'{safe_disease_name}_{safe_col}_患病比例.png')
            plt.close()
            print(f"{col}患病比例图已保存：{safe_disease_name}_{safe_col}_患病比
例.png")
```

```
# 执行描述性分析
descriptive_analysis(stroke_df, '中风', stroke_target, stroke_maps)
descriptive_analysis(heart_df, '心脏病', heart_target, heart_maps)
descriptive_analysis(cirrhosis_df, '肝硬化', cirrhosis_target, cirrhosis_maps)


# ----------------------
# 3. 差异性检验
# ----------------------
def difference_test(df, disease_name, target_col):
    """差异性检验，增加统计显著性解读"""
    print(f"\n===== {disease_name}差异性检验 =====")
    df_temp = df.copy()
    df_temp[target_col] = df_temp[target_col].map({'是': 1, '否': 0})
    sick_group = df_temp[df_temp[target_col] == 1]
    healthy_group = df_temp[df_temp[target_col] == 0]

    numeric_cols = df.select_dtypes(include=['float64', 'int64']).columns.tolist()
    categorical_cols = [col for col in df.select_dtypes(include=['int64']).columns
                        if col != target_col and df[col].nunique() < 10]

    # 连续变量检验结果整理
    significant_numeric = []
    for col in numeric_cols:
        if col == target_col: continue
        stat, p_norm = stats.shapiro(df[col])
        if p_norm > 0.05:
            stat, p_val = stats.ttest_ind(sick_group[col], healthy_group[col],
equal_var=False)
            test_type = "T 检验"
        else:
            stat, p_val = stats.mannwhitneyu(sick_group[col], healthy_group[col])
            test_type = "Mann-Whitney U 检验"

        if p_val < 0.05:
            significant_numeric.append({
                '特征': col,
                '检验类型': test_type,
                'p 值': p_val,
                '患病组均值': sick_group[col].mean(),
                '健康组均值': healthy_group[col].mean(),
                '效应大小': abs(sick_group[col].mean() - healthy_group[col].mean())
/ df[col].std()
            })

    # 离散变量检验结果整理
```

```
    significant_cat = []
    for col in categorical_cols:
        contingency = pd.crosstab(df[col], df[target_col])
        if contingency.shape[0] > 1 and contingency.shape[1] > 1:
            stat, p_val, dof, expected = stats.chi2_contingency(contingency)
            if p_val < 0.05:
                # 计算 Cramer's V 效应大小
                n = contingency.sum().sum()
                v = np.sqrt(stat / (n * (min(contingency.shape) - 1)))
                significant_cat.append({
                    '特征': col,
                    '检验类型': "卡方检验",
                    'p 值': p_val,
                    '效应大小': v,
                    '患病组占比': df[df[target_col] == '是'][col].value_counts(normalize=True).max()
                })

    # 结果输出与解读
    print("\n【显著差异特征】（p<0.05）")
    if significant_numeric:
        print("\n 连续变量：")
        for item in sorted(significant_numeric, key=lambda x: x['p 值'])[:3]:
            print(f"- {item['特征']}（{item['检验类型']}）：p={item['p 值']:.3f}，"
                  f"效应大小={item['效应大小']:.2f}，"
                  f"患病组均值{item['患病组均值']:.2f} vs 健康组{item['健康组均值']:.2f}")

    if significant_cat:
        print("\n 离散变量：")
        for item in sorted(significant_cat, key=lambda x: x['p 值'])[:3]:
            print(f"- {item['特征']}（{item['检验类型']}）：p={item['p 值']:.3f}，"
                  f"效应大小={item['效应大小']:.2f}，"
                  f"最高患病亚组占比{item['患病组占比']:.2%}")

    return significant_numeric + significant_cat


# 执行差异性检验并保存结果
stroke_significant = difference_test(stroke_df, '中风', stroke_target)
heart_significant = difference_test(heart_df, '心脏病', heart_target)
cirrhosis_significant = difference_test(cirrhosis_df, '肝硬化', cirrhosis_target)


# ----------------------
# 4. 相关性分析
# ----------------------
```

```
def cross_dataset_analysis(stroke_df, heart_df, cirrhosis_df, stroke_maps, heart_maps, cirrhosis_maps):
    """跨数据集对比分析，修复高血压列不存在的错误"""
    print(f"\n===== 跨数据集对比分析 =====")

    # 各数据集目标列
    stroke_target = '是否中风'
    heart_target = '是否有心脏病'
    cirrhosis_target = '是否肝硬化'

    # 1. 高血压患病率对比
    # 中风数据集
    stroke_hypertension = 0.0
    if '是否高血压' in stroke_df.columns:
        stroke_hypertension = stroke_df[stroke_df['是否高血压'] == 1][stroke_target].value_counts(normalize=True).get(
            '是', 0) * 100

    # 心脏病数据集
    heart_hypertension = 0.0
    if '是否高血压' in heart_df.columns:
        heart_hypertension = heart_df[heart_df['是否高血压'] == 1][heart_target].value_counts(normalize=True).get('是',

0) * 100
    else:
        print("警告：心脏病数据集中未找到'是否高血压'列，相关统计值记为 0")

    # 肝硬化数据集
    cirrhosis_hypertension = 0.0
    print("警告：肝硬化数据集中未找到'是否高血压'列，相关统计值记为 0")

    # 2. 平均年龄对比
    stroke_age = stroke_df[stroke_df[stroke_target] == '是']['年龄'].mean() if '年龄' in stroke_df.columns else 0
    heart_age = heart_df[heart_df[heart_target] == '是']['年龄'].mean() if '年龄' in heart_df.columns else 0
    cirrhosis_age_col = '年龄(年)' if '年龄(年)' in cirrhosis_df.columns else '年龄'
    cirrhosis_age = cirrhosis_df[cirrhosis_df[cirrhosis_target] == '是'][
        cirrhosis_age_col].mean() if cirrhosis_age_col in cirrhosis_df.columns else 0

    # 3. 高血糖患病率对比
    stroke_diabetes = stroke_df[stroke_df['是否高血糖'] == 1][stroke_target].value_counts(normalize=True).get('是',

0) * 100 if '是否高血糖' in stroke_df.columns else 0
```

```
    heart_diabetes = 0.0
    if '是否高血糖' in heart_df.columns:
        heart_diabetes = heart_df[heart_df['是否高血糖'] ==
1][heart_target].value_counts(normalize=True).get('是',

0) * 100
    else:
        print("警告：心脏病数据集中未找到'是否高血糖'列，相关统计值记为 0")

    # 输出共同风险因素对比
    print("\n【共同风险因素对比】")
    print(
        f"高血压患病率：中风{stroke_hypertension:.1f}%，心脏病
{heart_hypertension:.1f}%，肝硬化{cirrhosis_hypertension:.1f}%")
    print(f"患者平均年龄：中风{stroke_age:.1f}岁，心脏病{heart_age:.1f}岁，肝硬
化{cirrhosis_age:.1f}岁")
    print(f"高血糖患病率：中风{stroke_diabetes:.1f}%，心脏病
{heart_diabetes:.1f}%")

    # 4. 年龄分布对比
    plt.figure(figsize=(12, 6))
    if '年龄' in stroke_df.columns:
        sns.kdeplot(
            stroke_df[stroke_df[stroke_target] == '是']['年龄'],
            label=f'{stroke_target}-是',
            color='#e74c3c'
        )
        sns.kdeplot(
            stroke_df[stroke_df[stroke_target] == '否']['年龄'],
            label=f'{stroke_target}-否',
            color='#3498db'
        )
    if '年龄' in heart_df.columns:
        sns.kdeplot(
            heart_df[heart_df[heart_target] == '是']['年龄'],
            label=f'{heart_target}-是',
            color='#f39c12'
        )
        sns.kdeplot(
            heart_df[heart_df[heart_target] == '否']['年龄'],
            label=f'{heart_target}-否',
            color='#2ecc71'
        )
    if cirrhosis_age_col in cirrhosis_df.columns:
        sns.kdeplot(
            cirrhosis_df[cirrhosis_df[cirrhosis_target] == '是'][cirrhosis_age_col],
            label=f'{cirrhosis_target}-是',
```

```
            color='#9b59b6'
        )
        sns.kdeplot(
            cirrhosis_df[cirrhosis_df[cirrhosis_target] == '否'][cirrhosis_age_col],
            label=f'{cirrhosis_target}-否',
            color='#34495e'
        )

    plt.title('三种疾病患者与健康人群的年龄分布对比')
    plt.xlabel('年龄')
    plt.ylabel('密度')
    plt.legend(title='疾病状态', fontsize=9)
    plt.tight_layout()
    plt.savefig('三种疾病患者年龄分布对比.png')
    plt.close()
    print("三种疾病患者年龄分布对比图已保存")

    # 5. 性别差异分析
    plt.figure(figsize=(12, 6))
    bar_width = 0.25
    x = np.arange(2)  # 男/女两个类别位置

    # 中风数据集性别分布
    if '性别' in stroke_df.columns:
        stroke_gender = stroke_df.groupby('性别')[stroke_target].value_counts(normalize=True).unstack() * 100
        stroke_gender_map = {v: k for k, v in stroke_maps.get('性别', {}).items()}
        stroke_gender.index = [stroke_gender_map.get(idx, idx) for idx in stroke_gender.index]
        stroke_gender = stroke_gender.reindex(['男', '女'])
        plt.bar(x - bar_width, stroke_gender['是'], width=bar_width, label=stroke_target, color='#e74c3c')

    # 心脏病数据集性别分布
    if '性别' in heart_df.columns:
        heart_gender = heart_df.groupby('性别')[heart_target].value_counts(normalize=True).unstack() * 100
        heart_gender_map = {v: k for k, v in heart_maps.get('性别', {}).items()}
        heart_gender.index = [heart_gender_map.get(idx, idx) for idx in heart_gender.index]
        heart_gender = heart_gender.reindex(['男', '女'])
        plt.bar(x, heart_gender['是'], width=bar_width, label=heart_target, color='#f39c12')

    # 肝硬化数据集性别分布
    if '性别' in cirrhosis_df.columns:
        cirrhosis_gender = cirrhosis_df.groupby('性别
```

```
')[cirrhosis_target].value_counts(normalize=True).unstack() * 100
            cirrhosis_gender_map = {v: k for k, v in cirrhosis_maps.get('性别', {}).items()}
            cirrhosis_gender.index = [cirrhosis_gender_map.get(idx, idx) for idx in
cirrhosis_gender.index]
            cirrhosis_gender = cirrhosis_gender.reindex(['男', '女'])
            plt.bar(x + bar_width, cirrhosis_gender[' 是 '], width=bar_width,
label=cirrhosis_target, color='#9b59b6')

    plt.title('不同性别中的疾病患病率对比')
    plt.xlabel('性别')
    plt.ylabel('患病率 (%)')
    plt.xticks(x, ['男', '女'])
    plt.legend(title='疾病类型', fontsize=9)
    plt.tight_layout()
    plt.savefig('三种疾病患者性别分布对比.png')
    plt.close()
    print("三种疾病患者性别分布对比图已保存")

    # 6. 多疾病共同影响因素
    stroke_factors = [item['特征'] for item in stroke_significant]
    heart_factors = [item['特征'] for item in heart_significant]
    cirrhosis_factors = [item['特征'] for item in cirrhosis_significant]

    common_factors = []
    for factor in set(stroke_factors + heart_factors + cirrhosis_factors):
        count = 0
        diseases = []
        if factor in stroke_factors:
            count += 1
            diseases.append('中风')
        if factor in heart_factors:
            count += 1
            diseases.append('心脏病')
        if factor in cirrhosis_factors:
            count += 1
            diseases.append('肝硬化')
        if count >= 2:
            common_factors.append({'特征': factor, '影响疾病': diseases})

    print("\n【多疾病共同影响因素】")
    if common_factors:
        for f in common_factors:
            print(f"- {f['特征']}：影响{f['影响疾病']}")
    else:
        print("未发现两种及以上疾病共有的显著影响因素")
```

```
# 执行跨数据集对比分析
cross_dataset_analysis(stroke_df, heart_df, cirrhosis_df, stroke_maps, heart_maps, cirrhosis_maps)

print("\n===== 数据分析完成 =====")
print("所有图表已保存为 PNG 文件")
```

附录 3

介绍：该代码是 python 语言编写的，构建了一个疾病预测模型类，对中风、心脏病和肝硬化三个数据集进行预处理后，训练并评估逻辑回归、随机森林等多种分类模型，通过 ROC 曲线、特征重要性分析等优化模型，最终输出各疾病的最佳预测模型及性能指标。

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.ensemble import (RandomForestClassifier, GradientBoostingClassifier,
                              ExtraTreesClassifier, AdaBoostClassifier)
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import (accuracy_score, precision_score, recall_score, f1_score,
                             roc_auc_score, roc_curve, classification_report,
confusion_matrix)
from sklearn.feature_selection import SelectKBest, f_classif
from imblearn.over_sampling import SMOTE
import xgboost as xgb
import lightgbm as lgb
from sklearn.neural_network import MLPClassifier
import shap
from tqdm import tqdm
import os
import warnings

warnings.filterwarnings('ignore')

# 设置中文显示
plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False


class DiseasePredictionSystem:
    def __init__(self, data_paths):
        """初始化疾病预测系统，传入数据路径"""
        self.data_paths = data_paths
```

```
        # 扩展模型库：融合多个优势模型
        self.models = {
            'LogisticRegression': LogisticRegression(random_state=42,
max_iter=1000),
            'RandomForest': RandomForestClassifier(random_state=42),
            'GradientBoosting': GradientBoostingClassifier(random_state=42),
            'SVM': SVC(probability=True, random_state=42),
            'XGBoost': xgb.XGBClassifier(random_state=42, eval_metric='logloss',
verbosity=0),
            'LightGBM': lgb.LGBMClassifier(random_state=42, verbose=-1,
force_col_wise=True),
            'MLP': MLPClassifier(random_state=42, max_iter=1000),
            'KNeighbors': KNeighborsClassifier(),
            'ExtraTrees': ExtraTreesClassifier(random_state=42),
            'AdaBoost': AdaBoostClassifier(random_state=42)
        }
        self.best_models = {}
        self.scalers = {}
        self.results = {}
        self.feature_importances = {}
        self.label_encoders = {}
        self.training_history = {}
        self.shap_values = {}
    def load_and_preprocess(self):
        """加载并预处理所有疾病数据"""
        print("=== 数据预处理 ===")
        self.stroke_data = self._preprocess_stroke()
        self.heart_data = self._preprocess_heart()
        self.cirrhosis_data = self._preprocess_cirrhosis()
        print("数据预处理完成\n")

    def _preprocess_stroke(self):
        """预处理中风数据"""
        df = pd.read_csv(self.data_paths['stroke'])
        print(f"中风数据加载完成，共{len(df)}条记录")

        # 处理缺失值：数值型用中位数，分类特征用众数
        df['体重指数'] = df['体重指数'].replace("未知体重指数", np.nan)
        df['体重指数'] = pd.to_numeric(df['体重指数'])

        numeric_cols = ['年龄', '平均葡萄糖水平', '体重指数']
        for col in numeric_cols:
            df[col].fillna(df[col].median(), inplace=True)
            print(f"填充{col}缺失值 {df[col].isnull().sum()}个")

        categorical_cols = ['性别', '是否高血压', '是否心脏病', '是否结过婚',
                            '工作类型', '居住类型', '吸烟状态', '是否中风']
        for col in categorical_cols:
```

```
                df[col].fillna(df[col].mode()[0], inplace=True)
                print(f"填充{col}缺失值 {df[col].isnull().sum()}个")

        # 编码分类变量
        binary_cols = ['是否高血压', '是否心脏病', '是否结过婚', '是否中风']
        for col in binary_cols:
            df[col] = df[col].map({'是': 1, '否': 0})

        # 保存标签编码器
        le = LabelEncoder()
        df['性别_编码'] = le.fit_transform(df['性别'])
        self.label_encoders['stroke_gender'] = le

        le = LabelEncoder()
        df['工作类型_编码'] = le.fit_transform(df['工作类型'])
        self.label_encoders['stroke_work'] = le

        le = LabelEncoder()
        df['居住类型_编码'] = le.fit_transform(df['居住类型'])
        self.label_encoders['stroke_residence'] = le

        le = LabelEncoder()
        df['吸烟状态_编码'] = le.fit_transform(df['吸烟状态'])
        self.label_encoders['stroke_smoking'] = le

        # 选择特征
        features = ['性别_编码', '年龄', '是否高血压', '是否心脏病', '是否结过婚',
                    '工作类型_编码', '居住类型_编码', '平均葡萄糖水平', '体重指
数', '吸烟状态_编码']
        X = df[features]
        y = df['是否中风']

        # 处理类别不平衡
        print(f"中风类别分布：\n{y.value_counts()}")
        minority_ratio = y.mean()

        if minority_ratio < 0.1:  # 如果少数类占比低于 10%
            print("检测到严重数据不平衡，应用 SMOTE 过采样...")
            smote = SMOTE(random_state=42)
            X_resampled, y_resampled = smote.fit_resample(X, y)
            print(f"SMOTE 后类别分布：\n{y_resampled.value_counts()}")
            return {'X': X_resampled, 'y': y_resampled, 'feature_names': features,
'original_X': X, 'original_y': y}
        elif minority_ratio < 0.2:  # 如果少数类占比低于 20%
            print("检测到中度数据不平衡，将在模型训练中使用 class_weight 参
数...")
            return {'X': X, 'y': y, 'feature_names': features, 'needs_class_weight': True}
```

```
            else:
                return {'X': X, 'y': y, 'feature_names': features, 'needs_class_weight': False}

        def _preprocess_heart(self):
            """预处理心脏病数据"""
            df = pd.read_csv(self.data_paths['heart'])
            print(f"心脏病数据加载完成，共{len(df)}条记录")

            # 处理缺失值
            numeric_cols = ['年龄', '静息血压', '血清胆固醇', '最大心率', '抑郁症测量的数值']
            for col in numeric_cols:
                df[col] = pd.to_numeric(df[col], errors='coerce')
                df[col].fillna(df[col].median(), inplace=True)
                print(f"填充{col}缺失值  {df[col].isnull().sum()}个")

            categorical_cols = ['性别', '胸痛型', '空腹血糖是否大于 120 毫克/分升',
                                '静息心电图结果', '运动是否诱发心绞痛', '峰值运动 ST-T 波异常段的斜率', '是否有心脏病']
            for col in categorical_cols:
                df[col].fillna(df[col].mode()[0], inplace=True)
                print(f"填充{col}缺失值  {df[col].isnull().sum()}个")

            # 编码分类变量
            binary_cols = ['空腹血糖是否大于 120 毫克/分升', '运动是否诱发心绞痛', '是否有心脏病']
            for col in binary_cols:
                df[col] = df[col].map({'是': 1, '否': 0})

            # 保存标签编码器
            le = LabelEncoder()
            df['性别_编码'] = le.fit_transform(df['性别'])
            self.label_encoders['heart_gender'] = le

            le = LabelEncoder()
            df['胸痛型_编码'] = le.fit_transform(df['胸痛型'])
            self.label_encoders['heart_chest_pain'] = le

            le = LabelEncoder()
            df['静息心电图结果_编码'] = le.fit_transform(df['静息心电图结果'])
            self.label_encoders['heart_ecg'] = le

            le = LabelEncoder()
            df['峰值运动 ST-T 波异常段的斜率_编码'] = le.fit_transform(df['峰值运动 ST-T 波异常段的斜率'])
            self.label_encoders['heart_st_slope'] = le
```

```
        # 选择特征
        features = ['年龄', '性别_编码', '胸痛型_编码', '静息血压', '血清胆固醇',
                    '空腹血糖是否大于 120 毫克/分升', '静息心电图结果_编码', '
最大心率',
                    '运动是否诱发心绞痛', '抑郁症测量的数值', '峰值运动 ST-T
波异常段的斜率_编码']
        X = df[features]
        y = df['是否有心脏病']

        print(f"心脏病类别分布：\n{y.value_counts()}")
        # 检查是否需要类别权重
        minority_ratio = y.mean()
        needs_class_weight = minority_ratio < 0.2

        return {'X': X, 'y': y, 'feature_names': features, 'needs_class_weight':
needs_class_weight}

    def _preprocess_cirrhosis(self):
        """预处理肝硬化数据"""
        df = pd.read_csv(self.data_paths['cirrhosis'])
        print(f"肝硬化数据加载完成，共{len(df)}条记录")

        # 定义目标变量
        df['是否严重肝硬化'] = (df['疾病的组织学阶段(1、2、3 或 4)'] >= 3).astype(int)
        print(f"肝硬化严重程度分布：\n{df['是否严重肝硬化'].value_counts()}")

        # 处理缺失值
        numeric_cols = ['年龄(年)', '血清胆红素(毫克/分升)', '血清胆固醇(毫克/分
升)', '血蛋白(克/分升)',
                        '尿铜(微克/天)', '碱性磷酸酶(单位/升)', 'SGOT(单位/毫
升)',
                        '甘油三酯(毫克/分升)', '血小板(毫克/升)', '凝血酶原时间
(s)',
                        '疾病的组织学阶段(1、2、3 或 4)']
        for col in numeric_cols:
            df[col] = pd.to_numeric(df[col], errors='coerce')
            df[col].fillna(df[col].median(), inplace=True)
            print(f"填充{col}缺失值 {df[col].isnull().sum()}个")

        categorical_cols = ['性别', '药物类型', '是否存在腹水', '是否存在肝肿大',
                            '是否存在 Spiders', '是否存在水肿']
        for col in categorical_cols:
            df[col].fillna(df[col].mode()[0], inplace=True)
            print(f"填充{col}缺失值 {df[col].isnull().sum()}个")

        # 编码分类变量
```

```
        for col in ['是否存在腹水', '是否存在肝肿大', '是否存在 Spiders']:
            df[col + '_编码'] = df[col].map({'是': 1, '否': 0})

        # 保存标签编码器
        le = LabelEncoder()
        df['性别_编码'] = le.fit_transform(df['性别'])
        self.label_encoders['cirrhosis_gender'] = le

        le = LabelEncoder()
        df['药物类型_编码'] = le.fit_transform(df['药物类型'])
        self.label_encoders['cirrhosis_drug'] = le

        le = LabelEncoder()
        df['是否存在水肿_编码'] = le.fit_transform(df['是否存在水肿'])
        self.label_encoders['cirrhosis_edema'] = le

        # 选择特征
        features = ['年龄(年)', '性别_编码', '药物类型_编码', '是否存在腹水_编码',
                    '是否存在肝肿大_编码', '是否存在 Spiders_编码', '是否存在水肿_编码',
                    '血清胆红素(毫克/分升)', '血清胆固醇(毫克/分升)', '血小板(毫克/升)',
                    '疾病的组织学阶段(1、2、3 或 4)']
        X = df[features]
        y = df['是否严重肝硬化']

        print(f"严重肝硬化类别分布：\n{y.value_counts()}")
        # 检查是否需要类别权重
        minority_ratio = y.mean()
        needs_class_weight = minority_ratio < 0.2

        return {'X': X, 'y': y, 'feature_names': features, 'needs_class_weight': needs_class_weight}

    def train_and_evaluate(self):
        """训练并评估所有模型"""
        diseases = [
            ('stroke', '中风', self.stroke_data),
            ('heart', '心脏病', self.heart_data),
            ('cirrhosis', '肝硬化', self.cirrhosis_data)
        ]

        for data_key, disease_name, data in diseases:
            print(f"\n=== {disease_name}预测模型 ===")
            X, y = data['X'], data['y']
            X_train, X_test, y_train, y_test = train_test_split(
                X, y, test_size=0.2, random_state=42, stratify=y if data_key != 'stroke'
```

```
else None
                )

                # 数据标准化
                scaler = StandardScaler()
                X_train_scaled = scaler.fit_transform(X_train)
                X_test_scaled = scaler.transform(X_test)
                self.scalers[data_key] = scaler

                # 初始化训练历史
                self.training_history[data_key] = {}

                # 训练并评估所有模型
                eval_results = {}
                for model_name, model in tqdm(self.models.items(), desc=f"训练{disease_name}模型"):
                    print(f"\n 训练{model_name}...")
                    try:
                        # 处理类别不平衡
                        if data.get('needs_class_weight', False) and hasattr(model, 'class_weight'):
                            print(f"应用 class_weight='balanced'处理类别不平衡...")
                            model = model.__class__(**{**model.get_params(), 'class_weight': 'balanced'})

                        # 特殊处理支持训练历史的模型
                        history = None
                        if model_name == 'MLP':
                            # MLP 可以记录训练损失
                            model.fit(X_train_scaled, y_train)
                            history = model.loss_curve_
                            self.training_history[data_key][model_name] = {
                                'train_loss': history
                            }
                        elif model_name == 'XGBoost':
                            # XGBoost 可以使用 eval_set 记录评估结果
                            try:
                                eval_set = [(X_train, y_train), (X_test, y_test)]
                                model.fit(X_train, y_train, eval_set=eval_set,
                                          early_stopping_rounds=50,
verbose=False)
                                history = model.evals_result()
                                self.training_history[data_key][model_name] = {
                                    'train_logloss': history['validation_0']['logloss'],
                                    'val_logloss': history['validation_1']['logloss']
                                }
                            except TypeError:
                                # 不支持 early_stopping_rounds 参数时的处理
```

```
                model.fit(X_train, y_train)
            elif model_name == 'LightGBM':
                # LightGBM 也支持训练历史
                try:
                    eval_set = [(X_train, y_train), (X_test, y_test)]
                    model.fit(X_train, y_train, eval_set=eval_set,
                              early_stopping_rounds=50,
verbose=False)
                    history = model.evals_result_
                    self.training_history[data_key][model_name] = {
                        'train_logloss':
history['training']['binary_logloss'],
                        'val_logloss': history['valid_1']['binary_logloss']
                    }
                except TypeError:
                    # 不支持 early_stopping_rounds 参数时的处理
                    model.fit(X_train, y_train)
            elif model_name in ['GradientBoosting', 'AdaBoost']:
                # 梯度提升模型可以通过 staged_predict_proba 获取训练
过程
                n_estimators = model.n_estimators
                train_scores = []
                val_scores = []

                # 为支持训练进度显示，使用 tqdm 包装训练过程
                model.fit(X_train, y_train)

                # 计算每个阶段的性能
                if hasattr(model, 'staged_predict_proba'):
                    for i, y_pred in
enumerate(model.staged_predict_proba(X_train)):
                        train_scores.append(roc_auc_score(y_train,
y_pred[:, 1]))
                    for i, y_pred in
enumerate(model.staged_predict_proba(X_test)):
                        val_scores.append(roc_auc_score(y_test,
y_pred[:, 1]))

                    self.training_history[data_key][model_name] = {
                        'train_auc': train_scores,
                        'val_auc': val_scores
                    }
            elif model_name in ['LogisticRegression', 'SVM', 'KNeighbors',
'RandomForest', 'ExtraTrees']:
                # 为其他模型添加训练历史记录
                # 这些模型没有内置的训练过程跟踪，所以我们手动创
建一些数据点
                model.fit(X_train_scaled if model_name in ['SVM',
'KNeighbors', 'MLP'] else X_train, y_train)
```

```
                    # 模拟训练过程：在训练集和测试集上的性能
                    y_train_pred = model.predict_proba(
                        X_train_scaled if model_name in ['SVM',
'KNeighbors', 'MLP'] else X_train)[:, 1]
                    y_test_pred = model.predict_proba(
                        X_test_scaled if model_name in ['SVM', 'KNeighbors',
'MLP'] else X_test)[:, 1]

                    self.training_history[data_key][model_name] = {
                        'train_auc': [roc_auc_score(y_train, y_train_pred)],
                        'val_auc': [roc_auc_score(y_test, y_test_pred)]
                    }
                else:
                    # 其他模型正常训练
                    if model_name in ['SVM', 'KNeighbors', 'MLP']:
                        model.fit(X_train_scaled, y_train)
                    else:
                        model.fit(X_train, y_train)

                # 预测
                if model_name in ['SVM', 'KNeighbors', 'MLP']:
                    y_pred = model.predict(X_test_scaled)
                    y_prob = model.predict_proba(X_test_scaled)[:, 1] if
hasattr(model, 'predict_proba') else None
                else:
                    y_pred = model.predict(X_test)
                    y_prob = model.predict_proba(X_test)[:, 1] if
hasattr(model, 'predict_proba') else None

                # 计算评估指标
                if y_prob is not None:
                    auc = roc_auc_score(y_test, y_prob)
                else:
                    auc = np.nan

                eval_results[model_name] = {
                    'accuracy': accuracy_score(y_test, y_pred),
                    'precision': precision_score(y_test, y_pred),
                    'recall': recall_score(y_test, y_pred),
                    'f1': f1_score(y_test, y_pred),
                    'auc': auc,
                    'y_test': y_test,
                    'y_pred': y_pred,
                    'y_prob': y_prob,
                    'model': model
                }
                print(f"准确率: {eval_results[model_name]['accuracy']:.4f}")
                print(f"AUC: {eval_results[model_name]['auc']:.4f}" if not
```

```
np.isnan(auc) else "AUC: 不支持概率输出")

                        # 保存分类报告
                        report = classification_report(y_test, y_pred, target_names=['阴性', '阳性'])
                        eval_results[model_name]['report'] = report

                    except Exception as e:
                        print(f"模型 {model_name} 训练失败: {e}")
                        eval_results[model_name] = None

                # 可视化训练过程
                self._visualize_training_history(data_key, disease_name)

                # 选择最佳模型（基于 AUC）
                valid_models = {k: v for k, v in eval_results.items() if v is not None and not np.isnan(v['auc'])}
                if not valid_models:
                    print(f"未找到支持 AUC 评估的有效模型！")
                    continue

                best_model_name = max(valid_models, key=lambda k: valid_models[k]['auc'])
                self.best_models[data_key] = valid_models[best_model_name]
                self.results[data_key] = eval_results

                print(f"\n最佳模型: {best_model_name} (AUC: {valid_models[best_model_name]['auc']:.4f})")
                print(f"分类报告:\n{valid_models[best_model_name]['report']}")

                # 分析与可视化
                self._plot_roc_curve(valid_models[best_model_name], disease_name)
                self._plot_confusion_matrix(valid_models[best_model_name], disease_name)
                self._plot_prediction_distribution(valid_models[best_model_name], disease_name)
                self._analyze_feature_importance(X, y, disease_name, best_model_name,

valid_models[best_model_name]['model'])
                self._sensitivity_analysis(X, y, best_model_name, valid_models[best_model_name]['model'], disease_name,
                                           data_key)
                self._improve_model(X, y, disease_name, best_model_name, data_key)
                self._explain_model(X, y, best_model_name, valid_models[best_model_name]['model'], disease_name, data_key)
                print("-" * 70)

    def _visualize_training_history(self, data_key, disease_name):
```

```
        """可视化模型训练历史"""
        history = self.training_history[data_key]

        if not history:
            print(f"没有可用的训练历史数据用于{disease_name}模型")
            return

        plt.figure(figsize=(12, 8))
        plt.suptitle(f"{disease_name}模型训练过程")

        # 统计有训练历史的模型数量
        model_count = len(history)
        cols = 2
        rows = (model_count + cols - 1) // cols

        for i, (model_name, hist) in enumerate(history.items(), 1):
            plt.subplot(rows, cols, i)

            if 'train_loss' in hist:
                plt.plot(hist['train_loss'], label='训练损失')
                plt.ylabel('损失')
                plt.title(f'{model_name}训练损失')
            elif 'train_logloss' in hist:
                plt.plot(hist['train_logloss'], label='训练 LogLoss')
                plt.plot(hist['val_logloss'], label='验证 LogLoss')
                plt.ylabel('LogLoss')
                plt.title(f'{model_name}训练过程')
            elif 'train_auc' in hist:
                # 为所有模型添加训练历史可视化
                if len(hist['train_auc']) > 1:  # 多个训练点
                    plt.plot(hist['train_auc'], label='训练 AUC')
                    plt.plot(hist['val_auc'], label='验证 AUC')
                    plt.xlabel('迭代次数')
                else:  # 只有一个训练点的模型
                    plt.bar(['训练', '验证'], [hist['train_auc'][0], hist['val_auc'][0]],
alpha=0.7)
                    plt.ylim(0, 1.1)

                plt.ylabel('AUC')
                plt.title(f'{model_name}模型性能')

            plt.legend()
            plt.grid(True)

        plt.tight_layout()
        plt.subplots_adjust(top=0.9)
```

```
            # 保存图片
            save_path = r"F:\亚太杯\可视化"
            if not os.path.exists(save_path):
                os.makedirs(save_path)
            save_file = os.path.join(save_path, f"{disease_name}_训练历史.png")
            plt.savefig(save_file)
            print(f"训练历史图片已保存到: {save_file}")
            plt.show()

        def _analyze_feature_importance(self, X, y, disease_name, model_name, model):
            """分析特征重要性"""
            print(f"\n{disease_name}特征重要性分析")

            # 仅对树模型和 XGBoost/LightGBM 分析特征重要性
            if model_name in ['RandomForest', 'GradientBoosting', 'ExtraTrees', 'AdaBoost',
                              'DecisionTree', 'XGBoost', 'LightGBM']:
                feature_importance = pd.DataFrame({
                    '特征': X.columns,
                    '重要性': model.feature_importances_
                }).sort_values('重要性', ascending=False)

                self.feature_importances[disease_name] = feature_importance

                print("\nTop 10 重要特征:")
                print(feature_importance.head(10))

                # 可视化特征重要性
                plt.figure(figsize=(10, 6))
                sns.barplot(x='重要性', y='特征', data=feature_importance.head(10))
                plt.title(f'{disease_name}特征重要性（{model_name}）')
                plt.tight_layout()

                # 保存图片
                save_path = r"F:\亚太杯\可视化"
                if not os.path.exists(save_path):
                    os.makedirs(save_path)
                save_file = os.path.join(save_path, f"{disease_name}_{model_name}_特
征重要性.png")
                plt.savefig(save_file)
                print(f"特征重要性图片已保存到: {save_file}")
                plt.show()

        def _sensitivity_analysis(self, X, y, model_name, model, disease_name, data_key):
            """进行灵敏度分析和交叉验证"""
            print(f"\n{disease_name}模型灵敏度分析")

            # 交叉验证评估稳定性
```

```
            cv_scores = cross_val_score(model, X, y, cv=5, scoring='roc_auc')
            print(f"交叉验证 AUC: {cv_scores.mean():.4f} (±{cv_scores.std() * 2:.4f})")

            # 分析不同特征阈值下的性能变化
            if hasattr(model, 'predict_proba'):
                thresholds = np.linspace(0.1, 0.9, 9)
                precision_scores = []
                recall_scores = []
                f1_scores = []

                for threshold in thresholds:
                    X_train, X_test, y_train, y_test = train_test_split(
                        X, y, test_size=0.2, random_state=42
                    )

                    if model_name in ['SVM', 'KNeighbors', 'MLP']:
                        # 使用 data_key 而非 disease_name
                        model.fit(self.scalers[data_key].transform(X_train), y_train)
                        y_prob = model.predict_proba(self.scalers[data_key].transform(X_test))[:, 1]
                    else:
                        model.fit(X_train, y_train)
                        y_prob = model.predict_proba(X_test)[:, 1]

                    y_pred_adjusted = (y_prob >= threshold).astype(int)
                    precision = precision_score(y_test, y_pred_adjusted)
                    recall = recall_score(y_test, y_pred_adjusted)
                    f1 = f1_score(y_test, y_pred_adjusted)

                    precision_scores.append(precision)
                    recall_scores.append(recall)
                    f1_scores.append(f1)

                    print(f"阈值={threshold:.1f}: 精确率={precision:.4f}, 召回率={recall:.4f}, F1={f1:.4f}")

                # 绘制阈值-性能曲线
                plt.figure(figsize=(10, 6))
                plt.plot(thresholds, precision_scores, 'o-', label='精确率')
                plt.plot(thresholds, recall_scores, 's-', label='召回率')
                plt.plot(thresholds, f1_scores, 'd-', label='F1 分数')
                plt.xlabel('分类阈值')
                plt.ylabel('分数')
                plt.title(f'{disease_name}模型性能随分类阈值变化')
                plt.legend()
                plt.grid(True)
                plt.show()
```

```
                # 特征扰动测试
                print("\n 特征扰动测试:")
                if model_name not in ['SVM', 'KNeighbors']:
                    # 选择 Top 3 重要特征进行扰动测试
                    if disease_name in self.feature_importances:
                        top_features = self.feature_importances[disease_name][' 特 征
'][:3].tolist()
                        X_sample = X.sample(50, random_state=42)  # 选择部分样本
进行测试

                        for feature in top_features:
                            print(f"\n 对特征 '{feature}' 进行扰动测试:")
                            original_predictions = model.predict_proba(X_sample)[:, 1]

                            # 增加微小扰动
                            X_perturbed_up = X_sample.copy()
                            X_perturbed_up[feature] = X_perturbed_up[feature] * 1.05
# 增加 5%
                            perturbed_predictions_up =
model.predict_proba(X_perturbed_up)[:, 1]

                            # 减少微小扰动
                            X_perturbed_down = X_sample.copy()
                            X_perturbed_down[feature] = X_perturbed_down[feature]
* 0.95  # 减少 5%
                            perturbed_predictions_down =
model.predict_proba(X_perturbed_down)[:, 1]

                            # 计算平均变化
                            avg_change_up =
np.mean(np.abs(perturbed_predictions_up - original_predictions))
                            avg_change_down =
np.mean(np.abs(perturbed_predictions_down - original_predictions))

                            print(f" 增 加 5%: 预 测 概 率 平 均 变 化 =
{avg_change_up:.4f}")
                            print(f" 减 少 5%: 预 测 概 率 平 均 变 化 =
{avg_change_down:.4f}")

    def _improve_model(self, X, y, disease_name, best_model_name, data_key):
        """超参数调优和特征选择"""
        print(f"\n{disease_name}模型改进（{best_model_name}）")

        # 根据最佳模型类型设置调优参数
        # 扩展所有模型的超参数网格
        param_grids = {
            'RandomForest': {
                'n_estimators': [100, 200, 300],
```

```
            'max_depth': [None, 10, 20],
            'min_samples_split': [2, 5],
            'class_weight': [None, 'balanced']
        },
        'GradientBoosting': {
            'n_estimators': [100, 200],
            'learning_rate': [0.01, 0.1],
            'max_depth': [3, 5],
            'subsample': [0.8, 1.0]
        },
        'ExtraTrees': {
            'n_estimators': [100, 200, 300],
            'max_depth': [None, 10, 20],
            'min_samples_split': [2, 5],
            'class_weight': [None, 'balanced']
        },
        'AdaBoost': {
            'n_estimators': [50, 100, 200],
            'learning_rate': [0.01, 0.1]
        },
        'XGBoost': {
            'n_estimators': [100, 200],
            'learning_rate': [0.01, 0.1],
            'max_depth': [3, 5],
            'subsample': [0.8, 1.0],
            'colsample_bytree': [0.8, 1.0]
        },
        'LightGBM': {
            'n_estimators': [100, 200],
            'learning_rate': [0.01, 0.1],
            'max_depth': [3, 5],
            'subsample': [0.8, 1.0],
            'colsample_bytree': [0.8, 1.0]
        },
        'MLP': {
            'hidden_layer_sizes': [(100,), (100, 50), (50, 50)],
            'learning_rate_init': [0.001, 0.01],
            'alpha': [0.0001, 0.001]
        },
        'LogisticRegression': {
            'C': [0.01, 0.1, 1, 10],
            'penalty': ['l1', 'l2', 'elasticnet'],
            'solver': ['liblinear', 'saga'],
            'class_weight': [None, 'balanced']
        },
        'SVM': {
            'C': [0.1, 1, 10],
            'kernel': ['linear', 'rbf'],
            'gamma': ['scale']
        },
```

```
            'KNeighbors': {
                'n_neighbors': [3, 5, 7, 10],
                'weights': ['uniform', 'distance'],
                'p': [1, 2]
            }
        }

        # 获取对应模型的参数网格
        param_grid = param_grids.get(best_model_name, {})

        # 获取基础模型
        base_model = self.models[best_model_name]

        # 网格搜索调优
        print("开始超参数网格搜索...")
        # 针对 SVM 单独设置优化参数
        if best_model_name == 'SVM':
            # 启用缓存，并增加并行数
            grid_search = GridSearchCV(
                base_model,
                param_grid,
                cv=5,
                scoring='roc_auc',
                n_jobs=-1,
                verbose=1
            )
            # 训练前设置缓存（仅对 SVM 有效）
            base_model.set_params(cache_size=500)  # 500MB 缓存
        else:
            grid_search = GridSearchCV(base_model, param_grid, cv=5,
scoring='roc_auc', n_jobs=-1)

        grid_search.fit(X, y)
        print(f"最佳参数: {grid_search.best_params_}")
        print(f"调优后 AUC: {grid_search.best_score_:.4f}")

        # 特征选择
        print("开始特征选择...")
        selector = SelectKBest(f_classif, k=min(10, X.shape[1]))  # 最多选择 10 个
特征
        X_selected = selector.fit_transform(X, y)

        # 获取被选中的特征
        selected_mask = selector.get_support()
        selected_features = X.columns[selected_mask]
        print(f"选中的特征: {', '.join(selected_features)}")

        # 用选择的特征重新训练模型
```

```
            X_train, X_test, y_train, y_test = train_test_split(
                X_selected, y, test_size=0.2, random_state=42
            )

            best_model = grid_search.best_estimator_
            best_model.fit(X_train, y_train)
            y_prob = best_model.predict_proba(X_test)[:, 1]
            improved_auc = roc_auc_score(y_test, y_prob)
            print(f"特征选择后 AUC: {improved_auc:.4f}")

            # 保存改进后的模型
            self.best_models[data_key]['improved_model'] = best_model
            self.best_models[data_key]['selected_features'] = selected_features
            self.best_models[data_key]['improved_auc'] = improved_auc

        def _explain_model(self, X, y, model_name, model, disease_name, data_key):
            """使用 SHAP 值解释模型预测逻辑"""
            print(f"\n{disease_name}模型解释性分析")

            # 仅对支持 SHAP 的模型进行解释
            if model_name in ['RandomForest', 'GradientBoosting', 'ExtraTrees', 'XGBoost',
'LightGBM',
                              'LogisticRegression']:
                try:
                    # 创建 SHAP 解释器
                    if model_name in ['XGBoost', 'LightGBM', 'RandomForest',
'GradientBoosting', 'ExtraTrees']:
                        # 对树模型使用 TreeExplainer
                        explainer = shap.TreeExplainer(model)
                    else:
                        # 对线性模型使用 LinearExplainer
                        explainer = shap.LinearExplainer(model, X)

                    # 计算 SHAP 值
                    shap_values = explainer.shap_values(X)

                    # 保存 SHAP 值用于后续分析
                    self.shap_values[disease_name] = shap_values

                    # 可视化特征重要性（基于 SHAP 值）
                    plt.figure(figsize=(10, 6))
                    shap.summary_plot(shap_values, X, plot_type="bar", show=False)

                    feature_names = [col.replace("_编码", "") for col in X.columns]
                    shap.summary_plot(shap_values, X, plot_type="bar",
feature_names=feature_names, show=False)
                    plt.title(f'{disease_name}模型 SHAP 特征重要性')
                    plt.tight_layout()
```

```
                # 保存图片
                save_path = r"F:\亚太杯\可视化"
                if not os.path.exists(save_path):
                    os.makedirs(save_path)
                save_file = os.path.join(save_path, f"{disease_name}_SHAP 特征重要性.png")
                plt.savefig(save_file)
                print(f"SHAP 特征重要性图片已保存到: {save_file}")
                plt.show()

                # 可视化特征影响方向
                plt.figure(figsize=(10, 6))
                shap.summary_plot(shap_values, X, feature_names=feature_names, show=False)
                plt.title(f'{disease_name}模型 SHAP 特征影响')
                plt.tight_layout()

                save_file = os.path.join(save_path, f"{disease_name}_SHAP 特征影响.png")
                plt.savefig(save_file)
                print(f"SHAP 特征影响图片已保存到: {save_file}")
                plt.show()

                # 为前 3 个特征创建部分依赖图
                if hasattr(model, 'predict_proba'):
                    top_features = self.feature_importances[disease_name][' 特 征 '][:3].tolist()
                    for feature in top_features:
                        plt.figure(figsize=(10, 6))

                        # 计算部分依赖
                        values = np.linspace(X[feature].min(), X[feature].max(), 100)
                        pdp_values = []

                        for val in values:
                            X_copy = X.copy()
                            X_copy[feature] = val
                            if model_name in ['SVM', 'KNeighbors', 'MLP']:
                                pred = model.predict_proba(self.scalers[data_key].transform(X_copy))[:, 1]
                            else:
                                pred = model.predict_proba(X_copy)[:, 1]
                            pdp_values.append(np.mean(pred))

                        plt.plot(values, pdp_values)
                        plt.xlabel(feature.replace("_编码", ""))
```

```
                                plt.ylabel('平均预测概率')
                                plt.title(f'{disease_name}模型 {feature.replace("_编码",
"")} 的部分依赖图')
                                plt.grid(True)

                                save_file = os.path.join(save_path,
f"{disease_name}_{feature.replace('_编码', '')}_部分依赖图.png")
                                plt.savefig(save_file)
                                print(f"{feature.replace('_编码', '')}部分依赖图图片已保
存到: {save_file}")
                                plt.show()

                    except Exception as e:
                        print(f"SHAP 分析失败: {e}")
            else:
                print(f"{model_name}模型不支持 SHAP 解释，跳过模型解释性分析")

    def _plot_roc_curve(self, result, disease_name):
        """绘制 ROC 曲线"""
        fpr, tpr, _ = roc_curve(result['y_test'], result['y_prob'])
        plt.figure(figsize=(8, 6))
        plt.plot(fpr, tpr, label=f'ROC 曲线 (AUC = {result["auc"]:.4f})')
        plt.plot([0, 1], [0, 1], 'k--')
        plt.xlabel('假正率')
        plt.ylabel('真正率')
        plt.title(f'{disease_name}最佳模型 ROC 曲线')
        plt.legend()

        # 保存图片
        save_path = r"F:\亚太杯\可视化"
        if not os.path.exists(save_path):
            os.makedirs(save_path)
        save_file = os.path.join(save_path, f"{disease_name}_ROC 曲线.png")
        plt.savefig(save_file)
        print(f"ROC 曲线图片已保存到: {save_file}")
        plt.show()

    def _plot_confusion_matrix(self, result, disease_name):
        """绘制混淆矩阵"""
        cm = confusion_matrix(result['y_test'], result['y_pred'])

        plt.figure(figsize=(8, 6))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                    xticklabels=['阴性', '阳性'],
                    yticklabels=['阴性', '阳性'])
        plt.xlabel('预测标签')
```

```
        plt.ylabel('真实标签')
        plt.title(f'{disease_name}最佳模型混淆矩阵')

        # 保存图片
        save_path = r"F:\亚太杯\可视化"
        if not os.path.exists(save_path):
            os.makedirs(save_path)
        save_file = os.path.join(save_path, f"{disease_name}_混淆矩阵.png")
        plt.savefig(save_file)
        print(f"混淆矩阵图片已保存到: {save_file}")
        plt.show()

    def _plot_prediction_distribution(self, result, disease_name):
        """绘制预测结果分布"""
        y_test = result['y_test']
        y_prob = result['y_prob']

        plt.figure(figsize=(10, 6))

        # 绘制阴性样本的预测概率分布
        plt.hist(y_prob[y_test == 0], bins=20, alpha=0.5, label='阴性样本')

        # 绘制阳性样本的预测概率分布
        plt.hist(y_prob[y_test == 1], bins=20, alpha=0.5, label='阳性样本')

        plt.xlabel('预测概率')
        plt.ylabel('样本数量')
        plt.title(f'{disease_name}预测概率分布')
        plt.legend()
        plt.grid(True)

        # 保存图片
        save_path = r"F:\亚太杯\可视化"
        if not os.path.exists(save_path):
            os.makedirs(save_path)
        save_file = os.path.join(save_path, f"{disease_name}_预测概率分布.png")
        plt.savefig(save_file)
        print(f"预测概率分布图片已保存到: {save_file}")
        plt.show()

    def predict_patient_risk(self, disease_type, patient_data):
        """预测患者患病风险"""
        disease_key = disease_type.lower()[:3]  # 取前 3 个字符作为键

        if disease_key not in self.best_models:
            raise ValueError(f"未找到{disease_type}的模型，请先训练模型")
```

```
            model_info = self.best_models[disease_key]
            model = model_info.get('improved_model', model_info['model'])
            scaler = self.scalers[disease_key]

            # 检查是否使用了特征选择
            if 'selected_features' in model_info:
                selected_features = model_info['selected_features']
                patient_data_selected = patient_data[selected_features]
            else:
                patient_data_selected = patient_data

            # 标准化数据
            if isinstance(model, (SVC, KNeighborsClassifier, MLPClassifier)):
                patient_data_scaled = scaler.transform([patient_data_selected])
                risk_probability = model.predict_proba(patient_data_scaled)[0][1]
            else:
                risk_probability = model.predict_proba([patient_data_selected])[0][1]

            return risk_probability


# 主函数
if __name__ == "__main__":
    # 数据路径
    data_paths = {
        'stroke': r"F:\亚太杯\stroke.csv",
        'heart': r"F:\亚太杯\heart.csv",
        'cirrhosis': r"F:\亚太杯\cirrhosis.csv"
    }

    # 初始化并运行模型
    print("===== 疾病预测系统启动 =====")
    system = DiseasePredictionSystem(data_paths)
    system.load_and_preprocess()
    system.train_and_evaluate()
    print("===== 分析完成 =====")
```

附录 4

介绍：该代码是 python 语言编写的，构建了一个疾病分析器，对中风、心脏病和肝硬化数据集进行预处理后，通过多疾病风险概率估计、共同特征分布比较及可视化分析，实现多疾病关联与综合风险评估并输出相关图表。

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import LabelEncoder
import os
```

```
# 设置中文显示
plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False

# 定义特征名的中文映射
feature_chinese_mapping = {
    'gender_encoded': '性别',
    'Age': '年龄',
    'age': '年龄',
    'hypertension': '是否有高血压',
    'heart_disease': '是否有心脏病',
    'ever_married_encoded': '是否已婚',
    'work_type_encoded': '工作类型',
    'Residence_type_encoded': '居住类型',
    'avg_glucose_level': '平均葡萄糖水平',
    'bmi': '体重指数',
    'smoking_status_encoded': '吸烟状态',
    'Sex_encoded': '性别',
    'Drug_encoded': '药物类型',
    'Ascites_encoded': '是否存在腹水',
    'Hepatomegaly_encoded': '是否存在肝肿大',
    'Spiders_encoded': '是否存在蜘蛛痣',
    'Edema_encoded': '是否存在水肿',
    'Bilirubin': '血清胆红素',
    'Cholesterol': '血清胆固醇',
    'Albumin': '白蛋白',
    'Copper': '尿铜',
    'Alk_Phos': '碱性磷酸酶',
    'SGOT': '谷草转氨酶',
    'Tryglicerides': '甘油三酯',
    'Platelets': '每立方血小板',
    'Prothrombin': '凝血酶原时间',
    'Stage': '疾病的组织学阶段',
    'ChestPainType_encoded': '胸痛型',
    'RestingBP': '静息血压',
    'FastingBS': '空腹血糖',
    'RestingECG_encoded': '静息心电图结果',
    'MaxHR': '最大心率',
    'ExerciseAngina_encoded': '运动性心绞痛',
    'Oldpeak': '运动后 ST 段压低',
    'ST_Slope_encoded': 'ST 段斜率'
}


class DiseaseAnalyzer:
```

```
    def __init__(self):
        # 创建保存图片的文件夹
        self.save_folder = "E:\桌面\可视化"
        if not os.path.exists(self.save_folder):
            os.makedirs(self.save_folder)

    def load_and_preprocess_data(self):
        """加载和预处理数据"""
        print("===  数据加载和预处理  ===")

        # 加载数据
        try:
            self.stroke_data = pd.read_csv('stroke.csv')
            self.cirrhosis_data = pd.read_csv('cirrhosis.csv')
            self.heart_data = pd.read_csv('heart.csv')
            print("数据加载成功")
        except FileNotFoundError as e:
            print(f"文件未找到: {e}")
            return False

        # 预处理中风数据
        self.stroke_processed = self.preprocess_stroke_data()

        # 预处理肝硬化数据
        self.cirrhosis_processed = self.preprocess_cirrhosis_data()

        # 预处理心脏病数据
        self.heart_processed = self.preprocess_heart_data()

        print("数据预处理完成")
        return True

    def preprocess_stroke_data(self):
        """预处理中风数据"""
        data = self.stroke_data.copy()

        # 处理缺失值
        data['bmi'] = pd.to_numeric(data['bmi'], errors='coerce')
        data['bmi'] = data['bmi'].fillna(data['bmi'].median())

        # 编码分类变量
        le_gender = LabelEncoder()
        data['gender_encoded'] = le_gender.fit_transform(data['gender'])

        le_married = LabelEncoder()
        data['ever_married_encoded'] = le_married.fit_transform(data['ever_married'])

        le_work = LabelEncoder()
```

```
        data['work_type_encoded'] = le_work.fit_transform(data['work_type'])

        le_residence = LabelEncoder()
        data['Residence_type_encoded'] = le_residence.fit_transform(data['Residence_type'])

        le_smoking = LabelEncoder()
        data['smoking_status_encoded'] = le_smoking.fit_transform(data['smoking_status'])

        # 选择特征
        features = ['gender_encoded', 'age', 'hypertension', 'heart_disease',
                    'ever_married_encoded', 'work_type_encoded', 'Residence_type_encoded',
                    'avg_glucose_level', 'bmi', 'smoking_status_encoded']

        X = data[features]
        y = data['stroke']

        return {'X': X, 'y': y, 'feature_names': features}

    def preprocess_cirrhosis_data(self):
        """预处理肝硬化数据"""
        data = self.cirrhosis_data.copy()

        # 将年龄从以天为单位转换为以年为单位
        data['Age'] = data['Age'] / 365

        # 创建二分类目标变量（D 表示死亡，其他表示存活）
        data['cirrhosis_death'] = (data['Status'] == 'D').astype(int)

        # 打印缺失值统计信息，用于调试
        print("\n 肝硬化数据缺失值统计:")
        print(data.isnull().sum())

        # 处理数值列缺失值
        numeric_columns = ['Age', 'Bilirubin', 'Cholesterol', 'Albumin', 'Copper',
                           'Alk_Phos', 'SGOT', 'Tryglicerides', 'Platelets',
'Prothrombin', 'Stage']

        # 确保所有数值列都在数据中
        available_numeric = [col for col in numeric_columns if col in data.columns]

        # 使用中位数填充数值列缺失值
        imputer = SimpleImputer(strategy='median')
        data[available_numeric] = imputer.fit_transform(data[available_numeric])

        # 处理分类列缺失值
        categorical_columns = ['Sex', 'Drug', 'Ascites', 'Hepatomegaly', 'Spiders',
```

```
'Edema']

        # 为每个分类列创建缺失值指示列，并填充缺失值为最频繁值
        for col in categorical_columns:
            if col in data.columns:
                # 创建缺失值指示列
                data[col + '_missing'] = data[col].isnull().astype(int)
                data[col] = data[col].fillna(data[col].mode()[0])

        # 编码分类变量
        self.le_sex = LabelEncoder()  # 保存实例供后续使用
        data['Sex_encoded'] = self.le_sex.fit_transform(data['Sex'])

        le_drug = LabelEncoder()
        data['Drug_encoded'] = le_drug.fit_transform(data['Drug'])

        # 处理 Y/N 变量
        yn_columns = ['Ascites', 'Hepatomegaly', 'Spiders']
        for col in yn_columns:
            if col in data.columns:
                data[col + '_encoded'] = data[col].map({'Y': 1, 'N': 0})

        # 处理 Edema 变量
        edema_mapping = {'N': 0, 'S': 1, 'Y': 2}
        data['Edema_encoded'] = data['Edema'].map(edema_mapping)

        # 选择特征
        features = ['Age', 'Sex_encoded', 'Drug_encoded', 'Ascites_encoded',
                    'Hepatomegaly_encoded', 'Spiders_encoded', 'Edema_encoded',
                    'Bilirubin', 'Cholesterol', 'Albumin', 'Copper', 'Alk_Phos',
                    'SGOT', 'Trygliceride', 'Platelets', 'Prothrombin', 'Stage']

        # 确保所有特征都存在
        available_features = [f for f in features if f in data.columns]

        X = data[available_features]
        y = data['cirrhosis_death']

        # 验证处理后的数据是否还有缺失值
        print("\n 预处理后肝硬化数据缺失值统计:")
        missing_values = X.isnull().sum()
        print(missing_values)

        # 输出具体包含缺失值的列
        if missing_values.sum() > 0:
            print("\n 包含缺失值的列:")
            print(missing_values[missing_values > 0])
```

```
            # 确保没有缺失值
            assert X.isnull().sum().sum() == 0, f"预处理后的数据仍包含缺失值！缺失值
总数: {missing_values.sum()}"

            return {'X': X, 'y': y, 'feature_names': available_features}

        def preprocess_heart_data(self):
            """预处理心脏病数据"""
            data = self.heart_data.copy()

            # 编码分类变量
            le_sex = LabelEncoder()
            data['Sex_encoded'] = le_sex.fit_transform(data['Sex'])

            le_chest_pain = LabelEncoder()
            data['ChestPainType_encoded']                                    =
le_chest_pain.fit_transform(data['ChestPainType'])

            le_resting_ecg = LabelEncoder()
            data['RestingECG_encoded']                                       =
le_resting_ecg.fit_transform(data['RestingECG'])

            le_exercise_angina = LabelEncoder()
            data['ExerciseAngina_encoded']                                   =
le_exercise_angina.fit_transform(data['ExerciseAngina'])

            le_st_slope = LabelEncoder()
            data['ST_Slope_encoded'] = le_st_slope.fit_transform(data['ST_Slope'])

            # 选择特征
            features = ['Age', 'Sex_encoded', 'ChestPainType_encoded', 'RestingBP',
                        'Cholesterol', 'FastingBS', 'RestingECG_encoded', 'MaxHR',
                        'ExerciseAngina_encoded', 'Oldpeak', 'ST_Slope_encoded']

            X = data[features]
            y = data['HeartDisease']

            return {'X': X, 'y': y, 'feature_names': features}

        def multi_disease_analysis(self):
            """多疾病关联分析"""
            print("\n===  多疾病关联与综合风险评估  ===")

            # 创建综合数据集
            # 由于数据来源不同，我们需要创建一个统一的评估框架

            # 使用概率方法进行多疾病风险评估
            self.estimate_multi_disease_risk()
```

```
        # 共同特征分析
        self.analyze_common_features()

        # 深入共同特征分析
        self.analyze_deeper_common_features()

    def estimate_multi_disease_risk(self):
        """估计多疾病风险"""
        print("\n---  多疾病风险概率估计  ---")

        # 假设各疾病独立，计算联合概率

        # 获取各疾病的患病概率
        stroke_prob = self.stroke_processed['y'].mean()
        heart_prob = self.heart_processed['y'].mean()
        cirrhosis_prob = self.cirrhosis_processed['y'].mean()

        print(f"中风患病概率: {stroke_prob:.4f}")
        print(f"心脏病患病概率: {heart_prob:.4f}")
        print(f"肝硬化死亡概率: {cirrhosis_prob:.4f}")

        # 计算两种疾病同时发生的概率（假设独立）
        stroke_heart_prob = stroke_prob * heart_prob
        stroke_cirrhosis_prob = stroke_prob * cirrhosis_prob
        heart_cirrhosis_prob = heart_prob * cirrhosis_prob

        # 计算三种疾病同时发生的概率
        all_three_prob = stroke_prob * heart_prob * cirrhosis_prob

        print(f"\n 两种疾病同时发生的概率:")
        print(f"中风+心脏病: {stroke_heart_prob:.6f}")
        print(f"中风+肝硬化: {stroke_cirrhosis_prob:.6f}")
        print(f"心脏病+肝硬化: {heart_cirrhosis_prob:.6f}")
        print(f"\n 三种疾病同时发生的概率: {all_three_prob:.8f}")

        # 可视化多疾病风险概率
        diseases = ['中风', '心脏病', '肝硬化', '中风+心脏病', '中风+肝硬化', '心脏病+
肝硬化', '三种疾病']
        probabilities = [stroke_prob, heart_prob, cirrhosis_prob, stroke_heart_prob,
stroke_cirrhosis_prob,
                         heart_cirrhosis_prob, all_three_prob]

        plt.figure(figsize=(10, 6))
        plt.bar(diseases, probabilities)
        for i, v in enumerate(probabilities):
            plt.text(i, v, f'{v:.6f}', ha='center')
```

```
        plt.xlabel('疾病组合')
        plt.ylabel('患病概率')
        plt.title('多疾病风险概率')
        plt.xticks(rotation=45)
        plt.tight_layout()
        plt.savefig(os.path.join(self.save_folder, '多疾病风险概率.png'))
        plt.show()

    def analyze_common_features(self):
        """分析共同特征"""
        print("\n---  共同特征分析  ---")

        # 年龄分布比较
        plt.figure(figsize=(15, 5))

        plt.subplot(1, 3, 1)
        plt.hist(self.stroke_processed['X']['age'], bins=30, alpha=0.7, label='中风')
        plt.xlabel('年龄')
        plt.ylabel('频数')
        plt.title('中风患者年龄分布')
        plt.legend()

        plt.subplot(1, 3, 2)
        plt.hist(self.heart_processed['X']['Age'],  bins=30,  alpha=0.7,  label='心脏病',
color='orange')
        plt.xlabel('年龄')
        plt.ylabel('频数')
        plt.title('心脏病患者年龄分布')
        plt.legend()

        plt.subplot(1, 3, 3)
        plt.hist(self.cirrhosis_processed['X']['Age'], bins=30, alpha=0.7, label='肝硬化',
color='green')
        plt.xlabel('年龄')
        plt.ylabel('频数')
        plt.title('肝硬化患者年龄分布')
        plt.legend()

        plt.tight_layout()
        # 保存图片
        plt.savefig(os.path.join(self.save_folder, '多疾病年龄分布.png'))
        plt.show()

        # 性别分布比较
        plt.figure(figsize=(15, 5))

        # 中风数据性别分布
```

```
            plt.subplot(1, 3, 1)
            stroke_gender_counts = self.stroke_processed['X']['gender_encoded'].value_counts().sort_index()
            stroke_gender_labels = ['女性', '男性', '其他']
            stroke_gender_labels = stroke_gender_labels[:len(stroke_gender_counts)]
            bars = plt.bar(stroke_gender_labels, stroke_gender_counts)
            for bar in bars:
                height = bar.get_height()
                plt.text(bar.get_x() + bar.get_width() / 2., height,
                         f'{int(height)}', ha='center', va='bottom')
            plt.title('中风患者性别分布')

            # 心脏病数据性别分布
            plt.subplot(1, 3, 2)
            heart_gender_counts = self.heart_processed['X']['Sex_encoded'].value_counts().sort_index()
            heart_gender_labels = ['女性', '男性']
            heart_gender_labels = heart_gender_labels[:len(heart_gender_counts)]
            bars = plt.bar(heart_gender_labels, heart_gender_counts)
            for bar in bars:
                height = bar.get_height()
                plt.text(bar.get_x() + bar.get_width() / 2., height,
                         f'{int(height)}', ha='center', va='bottom')
            plt.title('心脏病患者性别分布')

            # 肝硬化数据性别分布
            plt.subplot(1, 3, 3)
            cirrhosis_gender_counts = self.cirrhosis_processed['X']['Sex_encoded'].value_counts().sort_index()
            if hasattr(self, 'le_sex') and hasattr(self.le_sex, 'classes_'):
                # 获取 LabelEncoder 的类别映射
                gender_mapping = {v: k for k, v in enumerate(self.le_sex.classes_)}
                # 反转映射并创建标签
                cirrhosis_gender_labels = ['女性' if code == gender_mapping.get('F', 0) else '男性' for code in
                                           cirrhosis_gender_counts.index]
            else:
                # 默认情况，以防 LabelEncoder 未正确保存
                cirrhosis_gender_labels = ['女性', '男性'][:len(cirrhosis_gender_counts)]

            bars = plt.bar(cirrhosis_gender_labels, cirrhosis_gender_counts)
            for bar in bars:
                height = bar.get_height()
                plt.text(bar.get_x() + bar.get_width() / 2., height,
                         f'{int(height)}', ha='center', va='bottom')
            plt.title('肝硬化患者性别分布')

            plt.tight_layout()
```

```
        plt.savefig(os.path.join(self.save_folder, '多疾病性别分布.png'))
        plt.show()

    def compare_common_numeric_features(self, features):
        """比较共同数值特征在不同疾病中的分布"""
        for feature in features:
            plt.figure(figsize=(15, 5))

            # 确保特征名称在不同数据集中的一致性
            feature_mapping = {
                'Age': {'stroke': 'age', 'heart': 'Age', 'cirrhosis': 'Age'},
                'avg_glucose_level': {'stroke': 'avg_glucose_level', 'heart': None,
'cirrhosis': None},
            }

            # 绘制各疾病特征分布
            plot_position = 1
            for disease, color, title in zip(
                    ['stroke', 'heart', 'cirrhosis'],
                    ['blue', 'orange', 'green'],
                    ['中风', '心脏病', '肝硬化']
            ):
                disease_feature = feature_mapping[feature][disease]
                if disease_feature and disease_feature in getattr(self,
f'{disease}_processed')['X'].columns:
                    plt.subplot(1, 3, plot_position)
                    plt.hist(getattr(self,
f'{disease}_processed')['X'][disease_feature], bins=30, alpha=0.7,
                             color=color)
                    plt.title(f'{title} 患 者
{feature_chinese_mapping.get(disease_feature, disease_feature)}分布')
                    plt.xlabel(feature_chinese_mapping.get(disease_feature,
disease_feature))
                    plt.ylabel('频数')
                    plot_position += 1

            plt.tight_layout()
            plt.savefig(os.path.join(self.save_folder, f'共同特征_{feature}_分布比
较.png'))
            plt.show()

    def analyze_deeper_common_features(self):
        """深入分析共同特征"""
        print("\n--- 深入共同特征分析 ---")

        # 找出三种疾病共有的特征
        stroke_features = set(self.stroke_processed['feature_names'])
        heart_features = set(self.heart_processed['feature_names'])
```

```
        cirrhosis_features = set(self.cirrhosis_processed['feature_names'])

        # 共同特征
        common_features =
stroke_features.intersection(heart_features).intersection(cirrhosis_features)

        # 中风与心脏病的共同特征
        stroke_heart_common = stroke_features.intersection(heart_features)

        # 中风与肝硬化的共同特征
        stroke_cirrhosis_common = stroke_features.intersection(cirrhosis_features)

        # 心脏病与肝硬化的共同特征
        heart_cirrhosis_common = heart_features.intersection(cirrhosis_features)

        print("\n 三种疾病的共同特征:")
        for feature in common_features:
            chinese_name = feature_chinese_mapping.get(feature, feature)
            print(f"- {feature} ({chinese_name})")

        print("\n 中风与心脏病的共同特征:")
        for feature in stroke_heart_common - common_features:
            chinese_name = feature_chinese_mapping.get(feature, feature)
            print(f"- {feature} ({chinese_name})")

        print("\n 中风与肝硬化的共同特征:")
        for feature in stroke_cirrhosis_common - common_features:
            chinese_name = feature_chinese_mapping.get(feature, feature)
            print(f"- {feature} ({chinese_name})")

        print("\n 心脏病与肝硬化的共同特征:")
        for feature in heart_cirrhosis_common - common_features:
            chinese_name = feature_chinese_mapping.get(feature, feature)
            print(f"- {feature} ({chinese_name})")

        # 可视化共同特征
        self.visualize_common_features(common_features)

        # 分析共同数值特征的分布
        self.analyze_common_numeric_features()

        # 明确列出可能的共同特征
        self.identify_potential_common_features()

    def identify_potential_common_features(self):
        """识别潜在的共同特征，处理特征名称不一致的情况"""
        print("\n---  潜在共同特征分析  ---")
```

```
        # 定义可能的共同特征及其在不同数据集中的名称
        potential_common = {
            '年龄': {
                'stroke': 'age',
                'heart': 'Age',
                'cirrhosis': 'Age'
            },
            '性别': {
                'stroke': 'gender',
                'heart': 'Sex',
                'cirrhosis': 'Sex'
            },
            '胆固醇': {
                'stroke': None,  # 中风数据集中没有胆固醇
                'heart': 'Cholesterol',
                'cirrhosis': 'Cholesterol'
            },
            '心脏病史': {
                'stroke': 'heart_disease',
                'heart': 'HeartDisease',
                'cirrhosis': None
            }
        }

        # 分析每种潜在共同特征
        for feature_name, datasets in potential_common.items():
            present_in = []
            for dataset, col_name in datasets.items():
                if col_name and col_name in getattr(self, f'{dataset}_processed')['X'].columns:
                    present_in.append(dataset)

            if len(present_in) >= 2:
                print(f"\n{feature_name} 存在于: {', '.join([d.title() for d in present_in])}")
                print("  数据列名:")
                for dataset in present_in:
                    col_name = datasets[dataset]
                    print(f"  - {dataset.title()}: {col_name} ({feature_chinese_mapping.get(col_name, col_name)})")

    def visualize_common_features(self, common_features):
        """可视化共同特征"""
        # 提取数值型共同特征
        numeric_common_features = []
        for feature in common_features:
            if 'encoded' not in feature and feature != 'Stage':  # 排除编码特征和阶段特征
```

```
                numeric_common_features.append(feature)

        if not numeric_common_features:
            print("没有找到共同的数值型特征")
            return

        # 为每个共同数值特征创建分布图
        for feature in numeric_common_features:
            plt.figure(figsize=(15, 5))

            # 中风数据
            if feature in self.stroke_processed['X'].columns:
                plt.subplot(1, 3, 1)
                plt.hist(self.stroke_processed['X'][feature], bins=30, alpha=0.7,
label='中风')
                plt.title(f'中风患者{feature_chinese_mapping.get(feature, feature)}
分布')
                plt.xlabel(feature_chinese_mapping.get(feature, feature))
                plt.ylabel('频数')
                plt.legend()

            # 心脏病数据
            if feature in self.heart_processed['X'].columns:
                plt.subplot(1, 3, 2)
                plt.hist(self.heart_processed['X'][feature], bins=30, alpha=0.7, label='
心脏病', color='orange')
                plt.title(f'心脏病患者{feature_chinese_mapping.get(feature,
feature)}分布')
                plt.xlabel(feature_chinese_mapping.get(feature, feature))
                plt.ylabel('频数')
                plt.legend()

            # 肝硬化数据
            if feature in self.cirrhosis_processed['X'].columns:
                plt.subplot(1, 3, 3)
                plt.hist(self.cirrhosis_processed['X'][feature], bins=30, alpha=0.7,
label='肝硬化', color='green')
                plt.title(f'肝硬化患者{feature_chinese_mapping.get(feature,
feature)}分布')
                plt.xlabel(feature_chinese_mapping.get(feature, feature))
                plt.ylabel('频数')
                plt.legend()

            plt.tight_layout()
            plt.savefig(os.path.join(self.save_folder, f'{feature}_分布比较.png'))
            plt.show()

    def analyze_common_numeric_features(self):
```

```
        """分析共同数值特征的分布"""
        print("\n--- 共同数值特征统计分析 ---")

        # 找出三种疾病共有的数值特征
        numeric_features = ['Age', 'Cholesterol']

        for feature in numeric_features:
            # 检查该特征是否存在于各疾病数据中
            stroke_exists = feature in self.stroke_processed['X'].columns
            heart_exists = feature in self.heart_processed['X'].columns
            cirrhosis_exists = feature in self.cirrhosis_processed['X'].columns

            if not (stroke_exists or heart_exists or cirrhosis_exists):
                continue

            print(f"\n 特征: {feature_chinese_mapping.get(feature, feature)}")

            # 计算各疾病的统计信息
            if stroke_exists:
                stroke_stats = self.stroke_processed['X'][feature].describe()
                print(
                    f"中风患者：均值={stroke_stats['mean']:.2f}，标准差={stroke_stats['std']:.2f}，最小值={stroke_stats['min']:.2f}，最大值={stroke_stats['max']:.2f}")

            if heart_exists:
                heart_stats = self.heart_processed['X'][feature].describe()
                print(
                    f"心脏病患者：均值={heart_stats['mean']:.2f}，标准差={heart_stats['std']:.2f}，最小值={heart_stats['min']:.2f}，最大值={heart_stats['max']:.2f}")

            if cirrhosis_exists:
                cirrhosis_stats = self.cirrhosis_processed['X'][feature].describe()
                print(
                    f"肝硬化患者：均值={cirrhosis_stats['mean']:.2f}，标准差={cirrhosis_stats['std']:.2f}，最小值={cirrhosis_stats['min']:.2f}，最大值={cirrhosis_stats['max']:.2f}")

            # 可视化血清胆固醇统计信息
            if feature == 'Cholesterol':
                self.visualize_cholesterol_stats(heart_stats, cirrhosis_stats)

    def visualize_cholesterol_stats(self, heart_stats, cirrhosis_stats):
        """可视化血清胆固醇统计信息"""
        diseases = ['心脏病', '肝硬化']
        means = [heart_stats['mean'], cirrhosis_stats['mean']]
        stds = [heart_stats['std'], cirrhosis_stats['std']]
```

```
            mins = [heart_stats['min'], cirrhosis_stats['min']]
            maxs = [heart_stats['max'], cirrhosis_stats['max']]

            x = np.arange(len(diseases))
            width = 0.2

            fig, ax = plt.subplots()
            rects1 = ax.bar(x - width, means, width, label='均值')
            rects2 = ax.bar(x, stds, width, label='标准差')
            rects3 = ax.bar(x + width, mins, width, label='最小值')
            rects4 = ax.bar(x + 2 * width, maxs, width, label='最大值')

            ax.set_ylabel('血清胆固醇值')
            ax.set_title('心脏病和肝硬化患者血清胆固醇统计信息')
            ax.set_xticks(x + width)
            ax.set_xticklabels(diseases)
            ax.legend()

            def autolabel(rects):
                for rect in rects:
                    height = rect.get_height()
                    ax.annotate('{:.2f}'.format(height),
                                xy=(rect.get_x() + rect.get_width() / 2, height),
                                xytext=(0, 3),
                                textcoords="offset points",
                                ha='center', va='bottom')

            autolabel(rects1)
            autolabel(rects2)
            autolabel(rects3)
            autolabel(rects4)

            fig.tight_layout()
            plt.savefig(os.path.join(self.save_folder, '血清胆固醇统计信息可视化.png'))
            plt.show()

        def run_analysis(self):
            if not self.load_and_preprocess_data():
                return
            self.multi_disease_analysis()
            print("\n===  分析完成  ===")


# 使用示例
if __name__ == "__main__":
    # 创建分析器实例
    analyzer = DiseaseAnalyzer()
```

```
    # 运行分析
    analyzer.run_analysis()
```