# python 3.8

import os

import sys

import platform

import random

import datetime

import pickle

import math

import json

import hashlib

import pandas as pd

import numpy as np

import matplotlib

# -------------------

GLOBAL\_SEED = 123 # 你指定的全局种子

os.environ["PYTHONHASHSEED"] = str(GLOBAL\_SEED)

random.seed(GLOBAL\_SEED)

np.random.seed(GLOBAL\_SEED)

matplotlib.use('AGG')

import matplotlib.pyplot as plt

from sklearn.base import clone, BaseEstimator, TransformerMixin

from sklearn.preprocessing import StandardScaler

from sklearn.inspection import permutation\_importance

from sklearn.model\_selection import cross\_val\_predict

from sklearn.model\_selection import cross\_validate

from sklearn.model\_selection import train\_test\_split as TTS

from sklearn.model\_selection import KFold, StratifiedKFold

from sklearn.model\_selection import GridSearchCV

from sklearn.pipeline import Pipeline

# 核心5种机器学习算法

from sklearn.linear\_model import LogisticRegression

from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier

from sklearn.tree import DecisionTreeClassifier

from xgboost import XGBClassifier

# 生存分析相关

from lifelines import KaplanMeierFitter

from lifelines import CoxPHFitter

import AnalysisFunction.X\_5\_SmartPlot as x5

from AnalysisFunction.X\_5\_SmartPlot import plot\_calibration\_curve

from AnalysisFunction.X\_5\_SmartPlot import calculate\_net\_benefit

from AnalysisFunction.X\_5\_SmartPlot import plot\_decision\_curves

from AnalysisFunction.X\_1\_DataGovernance import data\_standardization

from AnalysisFunction.X\_1\_DataGovernance import \_analysis\_dict

from AnalysisFunction.X\_2\_DataSmartStatistics import comprehensive\_smart\_analysis

from AnalysisFunction.utils\_ml import filtering, dic2str, round\_dec, save\_fig

from AnalysisFunction.utils\_ml import classification\_metric\_evaluate

from AnalysisFunction.utils\_ml import make\_class\_metrics\_dict

from AnalysisFunction.utils\_ml import ci

from sklearn.preprocessing import label\_binarize

from sklearn.metrics import roc\_auc\_score, brier\_score\_loss, roc\_curve

import shap

from functools import reduce

plt.rcParams['font.sans-serif'] = ['SimHei']

plt.rcParams['axes.unicode\_minus'] = False

from matplotlib import rc

plt.rcParams['ps.useafm'] = True

rc('font', \*\*{'family': 'sans-serif', 'sans-serif': ['FreeSans']})

plt.rcParams['pdf.fonttype'] = 42

# ================== 计数过程构造 ==================

def build\_counting\_process(

df: pd.DataFrame,

time\_col: str,

event\_col: str,

id\_col: str = None,

exposure\_intervals: dict = None,

same\_day\_event\_as\_preop: bool = True

) -> pd.DataFrame:

"""

将 (time, event) 生存数据构造成 counting-process 长表 (t\_start, t\_stop, status, exposure)。

- 如果未提供 exposure\_intervals，则每个样本生成单行：t\_start=0, t\_stop=time, status=event, exposure=0

- 如果提供 exposure\_intervals（形如 {id: [(s1,e1,exp1), (s2,e2,exp2), ...]}），

则依据这些区间切分，并在终止区间落点处赋 status。

- 同日手术与事件：若某区间起点与事件时间相同且 same\_day\_event\_as\_preop=True，

则事件计入“手术前”区间（即落在上一段区间，或在无上一段时落在 [0, t\_event) 区间）。

参数

----

df : 包含至少 time\_col, event\_col（以及可选 id\_col）的 DataFrame

time\_col : 生存时间（数值型，单位可自定，需与区间一致）

event\_col: 结局指示（1=事件，0=删失）

id\_col : 个体ID列名；若 None，则使用 df.index 作为 ID

exposure\_intervals : dict，给出每个ID的暴露区间列表：(t\_start, t\_stop, exposure\_flag/int/str)

same\_day\_event\_as\_preop : bool，同日事件按术前计入

返回

----

long\_df : DataFrame，列含 ['id','t\_start','t\_stop','status','exposure']

"""

if id\_col is None:

ids = df.index

else:

ids = df[id\_col]

out\_rows = []

for i, row in df.iterrows():

pid = row[id\_col] if id\_col else i

T = float(row[time\_col])

D = int(row[event\_col])

# 默认无暴露：单行

if exposure\_intervals is None or pid not in exposure\_intervals:

out\_rows.append(

{"id": pid, "t\_start": 0.0, "t\_stop": T, "status": int(D), "exposure": 0}

)

continue

# 存在暴露区间

intervals = sorted(exposure\_intervals[pid], key=lambda x: (float(x[0]), float(x[1])))

# 先构建不超过 T 的区间

cur\_start = 0.0

for (s, e, expv) in intervals:

s = float(s); e = float(e)

if s >= T: # 后续区间都在事件之后，无需加入

break

# 若两个区间之间存在空段，添加空暴露区间

if s > cur\_start:

seg\_stop = min(s, T)

out\_rows.append({"id": pid, "t\_start": cur\_start, "t\_stop": seg\_stop,

"status": 0, "exposure": 0})

cur\_start = seg\_stop

# 添加暴露区间

seg\_stop = min(e, T)

out\_rows.append({"id": pid, "t\_start": cur\_start, "t\_stop": seg\_stop,

"status": 0, "exposure": expv})

cur\_start = seg\_stop

if cur\_start >= T:

break

# 尾段（到 T）

if cur\_start < T:

out\_rows.append({"id": pid, "t\_start": cur\_start, "t\_stop": T,

"status": 0, "exposure": 0})

# 在事件落点处设置 status

if D == 1:

# 找到包含 T 的段（左闭右开 [t\_start, t\_stop)；终止时刻归前一段）

# 若存在某段 s==T（同日手术），并且 same\_day\_event\_as\_preop=True，则把事件放在上一段

idx\_target = None

for j in range(len(out\_rows)-1, -1, -1):

if out\_rows[j]["id"] != pid:

continue

s = out\_rows[j]["t\_start"]; e = out\_rows[j]["t\_stop"]

# 常规：T 落在该段右端点（e==T），事件记在该段

if np.isclose(e, T):

idx\_target = j

# 如果段的起点也与 T 相同，说明同日手术起始于 T

if same\_day\_event\_as\_preop and np.isclose(s, T):

# 尝试将事件计入上一段

# 找到上一段（同一id且紧邻）

for k in range(j-1, -1, -1):

if out\_rows[k]["id"] == pid:

idx\_target = k

break

break

if idx\_target is not None:

out\_rows[idx\_target]["status"] = 1

long\_df = pd.DataFrame(out\_rows)

return long\_df

# ================== Cox 特征选择变换器 ==================

class CoxSelector(BaseEstimator, TransformerMixin):

"""

在每个CV训练折中进行：

1) 单因素 Cox 回归（p < p\_univ 进入候选）

2) 多因素 Cox 回归（最终保留 p < p\_multiv 的变量）

注意：严格使用 fit(X\_train\_fold) 的索引来子集生存数据，避免信息泄露。

"""

def \_\_init\_\_(self,

survival\_times\_full: pd.Series,

event\_indicators\_full: pd.Series,

p\_univ: float = 0.10,

p\_multiv: float = 0.05,

max\_vars: int = None,

penalizer: float = 0.0,

robust: bool = True):

self.survival\_times\_full = survival\_times\_full

self.event\_indicators\_full = event\_indicators\_full

self.p\_univ = p\_univ

self.p\_multiv = p\_multiv

self.max\_vars = max\_vars

self.penalizer = penalizer

self.robust = robust

self.selected\_features\_ = None

def \_safe\_cox\_fit(self, df, cols):

"""对 lifelines CoxPHFitter 做鲁棒拟合，必要时回退到带惩罚或去除共线。"""

cph = CoxPHFitter(penalizer=self.penalizer)

try:

cph.fit(df[['\_\_time\_\_', '\_\_event\_\_'] + cols],

duration\_col='\_\_time\_\_', event\_col='\_\_event\_\_', robust=self.robust)

return cph

except Exception:

cph = CoxPHFitter(penalizer=max(self.penalizer, 0.1))

cph.fit(df[['\_\_time\_\_', '\_\_event\_\_'] + cols],

duration\_col='\_\_time\_\_', event\_col='\_\_event\_\_', robust=True)

return cph

def fit(self, X: pd.DataFrame, y=None):

if not isinstance(X, pd.DataFrame):

raise ValueError("CoxSelector 需要带索引的 pandas.DataFrame")

# 只取当前训练折的索引对应的生存信息

idx = X.index

t = self.survival\_times\_full.loc[idx]

e = self.event\_indicators\_full.loc[idx]

# 组装用于 lifelines 的数据框

df = pd.DataFrame({'\_\_time\_\_': t, '\_\_event\_\_': e}).join(X)

# ---------- 单因素 Cox ----------

univ\_keep = []

for col in X.columns:

try:

cph\_u = CoxPHFitter(penalizer=self.penalizer)

cph\_u.fit(df[['\_\_time\_\_', '\_\_event\_\_', col]], duration\_col='\_\_time\_\_', event\_col='\_\_event\_\_', robust=self.robust)

pval = cph\_u.summary.loc[col, 'p']

if np.isfinite(pval) and pval < self.p\_univ:

univ\_keep.append(col)

except Exception:

continue

if len(univ\_keep) == 0:

univ\_keep = list(X.columns)

# 可选：限制候选数量

if self.max\_vars is not None and len(univ\_keep) > self.max\_vars:

pvals = []

for col in univ\_keep:

try:

cph\_u = CoxPHFitter(penalizer=self.penalizer)

cph\_u.fit(df[['\_\_time\_\_', '\_\_event\_\_', col]], duration\_col='\_\_time\_\_', event\_col='\_\_event\_\_', robust=self.robust)

pvals.append((col, float(cph\_u.summary.loc[col, 'p'])))

except Exception:

pvals.append((col, np.inf))

pvals\_sorted = sorted(pvals, key=lambda x: x[1])

univ\_keep = [c for c, \_ in pvals\_sorted[:self.max\_vars]]

# ---------- 多因素 Cox ----------

final\_keep = []

try:

cph\_m = self.\_safe\_cox\_fit(df, univ\_keep)

summ = cph\_m.summary

final\_keep = [var for var in univ\_keep

if var in summ.index and np.isfinite(summ.loc[var, 'p']) and summ.loc[var, 'p'] < self.p\_multiv]

except Exception:

final\_keep = univ\_keep

if len(final\_keep) == 0:

pvals = []

for col in univ\_keep:

try:

cph\_u = CoxPHFitter(penalizer=self.penalizer)

cph\_u.fit(df[['\_\_time\_\_', '\_\_event\_\_', col]], duration\_col='\_\_time\_\_', event\_col='\_\_event\_\_', robust=self.robust)

pvals.append((col, float(cph\_u.summary.loc[col, 'p'])))

except Exception:

pvals.append((col, np.inf))

pvals\_sorted = sorted(pvals, key=lambda x: x[1])

final\_keep = [c for c, \_ in pvals\_sorted[:max(1, min(3, len(pvals\_sorted)))]]

self.selected\_features\_ = final\_keep

return self

def transform(self, X: pd.DataFrame):

if self.selected\_features\_ is None:

raise RuntimeError("CoxSelector 尚未 fit")

cols = [c for c in self.selected\_features\_ if c in X.columns]

if len(cols) == 0:

return X

return X[cols]

# ==================（稳健化）==================

def calculate\_ipcw\_weights(survival\_times, event\_indicators, prediction\_horizon, eps: float = 1e-6):

"""

计算 IPCW 权重（基于删失分布 G(t) 的 Kaplan-Meier 估计）

记 C 为删失时间，则 G(t) = P(C >= t)。权重规则：

- 若个体在 horizon 之前发生事件：w\_i = 1 / G(t\_i-)

- 若个体在 horizon 之后（或未在 horizon 前发生事件）：w\_i = 1 / G(horizon)

- 若个体在 horizon 前被删失：该样本通常不用于 AUC 贡献（在标签构造中已被过滤）

为避免数值不稳定，这里使用下界截断：G(t) = max(G(t), eps)

"""

kmf = KaplanMeierFitter()

censoring\_indicators = 1 - np.asarray(event\_indicators, dtype=int)

st = np.asarray(survival\_times, dtype=float)

try:

kmf.fit(st, event\_observed=censoring\_indicators)

# 预先计算 G(t) 在关键点的值，减少重复查询

G\_h = float(kmf.survival\_function\_at\_times(prediction\_horizon).values[0])

G\_h = max(G\_h, eps)

weights = np.zeros\_like(st, dtype=float)

for i, (t, delta) in enumerate(zip(st, event\_indicators)):

if t <= prediction\_horizon:

if delta == 1:

G\_t = float(kmf.survival\_function\_at\_times(t).values[0])

G\_t = max(G\_t, eps)

weights[i] = 1.0 / G\_t

else:

# 在标签构造中，这类个体不会进入有效索引；给0以示占位

weights[i] = 0.0

else:

weights[i] = 1.0 / G\_h

except Exception as e:

print(f"IPCW计算错误: {e}")

weights = np.ones(len(survival\_times))

return weights

def create\_binary\_labels\_multiple\_horizons(survival\_times, event\_indicators, horizons):

"""

为多个预测时间点创建二分类标签和IPCW权重。

规则：

- 若 t\_i <= h 且 delta\_i==1 -> label=1（事件发生）

- 若 t\_i > h -> label=0（在 h 时刻尚未事件）

- 若 t\_i <= h 且 delta\_i==0 -> 被删失，剔除（不纳入该 h 的评估）

"""

results\_dict = {}

st = np.asarray(survival\_times, dtype=float)

ev = np.asarray(event\_indicators, dtype=int)

for horizon in horizons:

labels = []

valid\_indices = []

for i, (t, delta) in enumerate(zip(st, ev)):

if t <= horizon:

if delta == 1:

labels.append(1)

valid\_indices.append(i)

else:

# 删失在h之前：不计入

continue

else:

labels.append(0)

valid\_indices.append(i)

weights = calculate\_ipcw\_weights(st, ev, horizon)

valid\_weights = weights[valid\_indices]

results\_dict[f'{horizon}year'] = {

'labels': np.array(labels, dtype=int),

'weights': valid\_weights,

'indices': valid\_indices

}

return results\_dict

def get\_ml\_algorithms():

"""

返回研究中使用的5种机器学习算法（统一随机种子）

"""

algorithms = {

'RandomForestClassifier': RandomForestClassifier(random\_state=GLOBAL\_SEED),

'LogisticRegression': LogisticRegression(random\_state=GLOBAL\_SEED, max\_iter=1000),

'XGBClassifier': XGBClassifier(random\_state=GLOBAL\_SEED, eval\_metric='logloss', n\_estimators=100),

'DecisionTreeClassifier': DecisionTreeClassifier(random\_state=GLOBAL\_SEED),

'GradientBoostingClassifier': GradientBoostingClassifier(random\_state=GLOBAL\_SEED)

}

return algorithms

def get\_hyperparameter\_grids():

"""

为5种算法定义详细的网格搜索参数

（保留你原来的网格；注意：部分 sklearn 版本中 'auto' 可能触发警告）

"""

param\_grids = {

'RandomForestClassifier': {

'n\_estimators': [50, 60, 70, 80, 90, 100, 110, 120, 150, 200, 300, 500],

'max\_depth': [3, 5, 6, 7, 8, 9, 10, 15, 20, None],

'min\_samples\_split': [2, 5, 10, 15, 20],

'min\_samples\_leaf': [1, 2, 4, 6, 8],

'max\_features': ['auto', 'sqrt', 'log2', 0.3, 0.5, 0.7],

'bootstrap': [True, False]

},

'LogisticRegression': {

'C': [0.001, 0.01, 0.1, 0.5, 1.0, 5.0, 10.0, 50.0, 100.0],

'penalty': ['l1', 'l2', 'elasticnet'],

'solver': ['liblinear', 'saga', 'lbfgs'],

'l1\_ratio': [0.1, 0.3, 0.5, 0.7, 0.9],

'class\_weight': [None, 'balanced']

},

'XGBClassifier': {

'n\_estimators': [50, 100, 200, 300, 500],

'max\_depth': [3, 4, 5, 6, 8, 10],

'learning\_rate': [0.01, 0.05, 0.1, 0.15, 0.2, 0.3],

'subsample': [0.6, 0.7, 0.8, 0.9, 1.0],

'colsample\_bytree': [0.6, 0.7, 0.8, 0.9, 1.0],

'gamma': [0, 0.1, 0.2, 0.3, 0.5],

'min\_child\_weight': [1, 2, 3, 4, 5],

'reg\_alpha': [0, 0.01, 0.1, 0.5, 1.0],

'reg\_lambda': [0, 0.01, 0.1, 0.5, 1.0]

},

'DecisionTreeClassifier': {

'max\_depth': [3, 5, 7, 10, 15, 20, 25, None],

'min\_samples\_split': [2, 5, 10, 15, 20, 25, 30],

'min\_samples\_leaf': [1, 2, 4, 6, 8, 10, 12],

'max\_features': ['auto', 'sqrt', 'log2', None, 0.3, 0.5, 0.7],

'criterion': ['gini', 'entropy'],

'splitter': ['best', 'random'],

'class\_weight': [None, 'balanced']

},

'GradientBoostingClassifier': {

'n\_estimators': [50, 100, 200, 300, 500],

'learning\_rate': [0.01, 0.05, 0.1, 0.15, 0.2, 0.3],

'max\_depth': [3, 4, 5, 6, 7, 8],

'min\_samples\_split': [2, 5, 10, 15, 20],

'min\_samples\_leaf': [1, 2, 4, 6, 8],

'max\_features': ['auto', 'sqrt', 'log2', None, 0.3, 0.5, 0.7],

'subsample': [0.6, 0.7, 0.8, 0.9, 1.0]

}

}

return param\_grids

def calculate\_weighted\_auc\_brier(y\_true, y\_pred\_proba, weights):

"""

计算IPCW加权的AU-ROC和Brier Score

"""

weights = np.asarray(weights, dtype=float)

if weights.sum() > 0:

weights = weights / np.sum(weights) \* len(weights)

try:

weighted\_auc = roc\_auc\_score(y\_true, y\_pred\_proba, sample\_weight=weights)

weighted\_brier = np.average((y\_pred\_proba - y\_true) \*\* 2, weights=weights)

except Exception as e:

print(f"加权指标计算错误: {e}")

weighted\_auc = roc\_auc\_score(y\_true, y\_pred\_proba)

weighted\_brier = brier\_score\_loss(y\_true, y\_pred\_proba)

return weighted\_auc, weighted\_brier

def split\_datasets(df\_seer, df\_chinese, features, survival\_col, event\_col, test\_ratio=0.2, random\_state=GLOBAL\_SEED):

"""

将SEER数据按8:2划分训练/内部测试集，中国医院数据作为外部测试集

"""

seer\_features = df\_seer[features].dropna()

seer\_survival = df\_seer[survival\_col]

seer\_events = df\_seer[event\_col]

valid\_indices = seer\_features.index

seer\_survival = seer\_survival.loc[valid\_indices]

seer\_events = seer\_events.loc[valid\_indices]

# Stratified 按事件分层，固定随机数

X\_train\_seer, X\_test\_seer, y\_survival\_train, y\_survival\_test, y\_event\_train, y\_event\_test = TTS(

seer\_features,

pd.concat([seer\_survival, seer\_events], axis=1),

test\_size=test\_ratio,

random\_state=random\_state,

stratify=seer\_events.loc[valid\_indices]

)

y\_survival\_train, y\_event\_train = y\_survival\_train.iloc[:, 0], y\_survival\_train.iloc[:, 1]

y\_survival\_test, y\_event\_test = y\_survival\_test.iloc[:, 0], y\_survival\_test.iloc[:, 1]

chinese\_features = df\_chinese[features].dropna()

chinese\_survival = df\_chinese[survival\_col].loc[chinese\_features.index]

chinese\_events = df\_chinese[event\_col].loc[chinese\_features.index]

data\_splits = {

'train': {

'features': X\_train\_seer,

'survival\_times': y\_survival\_train,

'event\_indicators': y\_event\_train

},

'internal\_test': {

'features': X\_test\_seer,

'survival\_times': y\_survival\_test,

'event\_indicators': y\_event\_test

},

'external\_test': {

'features': chinese\_features,

'survival\_times': chinese\_survival,

'event\_indicators': chinese\_events

}

}

return data\_splits

def \_prefix\_param\_grid(param\_grid, prefix):

"""把参数网格键加上前缀，例如 'n\_estimators' -> 'clf\_\_n\_estimators' """

return {f"{prefix}{k}": v for k, v in param\_grid.items()}

def perform\_grid\_search\_with\_cv(algorithm, param\_grid, X\_train, y\_train, sample\_weights,

cv=10, scoring='roc\_auc',

cox\_times\_full=None, cox\_events\_full=None,

cox\_p\_univ=0.10, cox\_p\_multiv=0.05, cox\_max\_vars=None,

cox\_penalizer=0.0, cox\_robust=True):

"""

执行网格搜索+ StratifiedKFold 交叉验证，并在每个训练折中进行 Cox 单/多因素特征筛选

"""

print(f"开始网格搜索 {algorithm.\_\_class\_\_.\_\_name\_\_}...")

print(f"参数搜索空间大小: {np.prod([len(v) for v in param\_grid.values()])} 种组合")

# 统一CV为 StratifiedKFold（可复现）

cv\_splitter = StratifiedKFold(n\_splits=cv, shuffle=True, random\_state=GLOBAL\_SEED)

# 构造包含 CoxSelector 的 Pipeline

cox\_selector = CoxSelector(

survival\_times\_full=cox\_times\_full,

event\_indicators\_full=cox\_events\_full,

p\_univ=cox\_p\_univ,

p\_multiv=cox\_p\_multiv,

max\_vars=cox\_max\_vars,

penalizer=cox\_penalizer,

robust=cox\_robust

)

# 注意：参数网格需要前缀到 'clf\_\_'

if algorithm.\_\_class\_\_.\_\_name\_\_ == 'LogisticRegression':

# 处理 Logistic 的 solver-penalty 约束，同时加 'clf\_\_' 前缀

valid\_param\_grid = []

for penalty in param\_grid['penalty']:

for solver in param\_grid['solver']:

if (penalty == 'l1' and solver in ['liblinear', 'saga']) or \

(penalty == 'l2' and solver in ['liblinear', 'saga', 'lbfgs']) or \

(penalty == 'elasticnet' and solver == 'saga'):

combo = {

'clf\_\_penalty': [penalty],

'clf\_\_solver': [solver],

'clf\_\_C': param\_grid['C'],

'clf\_\_class\_weight': param\_grid['class\_weight']

}

if penalty == 'elasticnet':

combo['clf\_\_l1\_ratio'] = param\_grid['l1\_ratio']

valid\_param\_grid.append(combo)

best\_score = -np.inf

best\_model = None

best\_params = None

for combo in valid\_param\_grid:

pipe = Pipeline([

('cox\_sel', cox\_selector),

('clf', clone(algorithm))

])

try:

grid\_search = GridSearchCV(

estimator=pipe,

param\_grid=combo,

cv=cv\_splitter,

scoring=scoring,

n\_jobs=-1,

verbose=0

)

# sample\_weight 需要以 'clf\_\_sample\_weight' 传入最后一步

grid\_search.fit(X\_train, y\_train, \*\*{'clf\_\_sample\_weight': sample\_weights})

if grid\_search.best\_score\_ > best\_score:

best\_score = grid\_search.best\_score\_

best\_model = grid\_search.best\_estimator\_

best\_params = grid\_search.best\_params\_

except Exception as e:

print(f"参数组合 {combo} 出错: {e}")

continue

cv\_scores = best\_score

else:

pipe = Pipeline([

('cox\_sel', cox\_selector),

('clf', algorithm)

])

grid\_search = GridSearchCV(

estimator=pipe,

param\_grid=\_prefix\_param\_grid(param\_grid, 'clf\_\_'),

cv=cv\_splitter,

scoring=scoring,

n\_jobs=-1,

verbose=1

)

grid\_search.fit(X\_train, y\_train, \*\*{'clf\_\_sample\_weight': sample\_weights})

best\_model = grid\_search.best\_estimator\_

best\_params = grid\_search.best\_params\_

cv\_scores = grid\_search.best\_score\_

print(f"最优参数: {best\_params}")

print(f"交叉验证最优分数: {cv\_scores:.4f}")

return best\_model, best\_params, cv\_scores

# ================== manifest 导出 ==================

def \_export\_manifest\_and\_grids(save\_dir: str, param\_grids: dict, cv\_splits: int):

os.makedirs(save\_dir, exist\_ok=True)

manifest = {

"python": sys.version,

"platform": platform.platform(),

"GLOBAL\_SEED": GLOBAL\_SEED,

"cv": {"type": "StratifiedKFold", "n\_splits": cv\_splits, "shuffle": True, "random\_state": GLOBAL\_SEED},

"packages": {

"pandas": pd.\_\_version\_\_,

"numpy": np.\_\_version\_\_,

"matplotlib": matplotlib.\_\_version\_\_,

"scikit\_learn": \_\_import\_\_("sklearn").\_\_version\_\_,

"xgboost": \_\_import\_\_("xgboost").\_\_version\_\_,

"lifelines": \_\_import\_\_("lifelines").\_\_version\_\_,

"shap": shap.\_\_version\_\_

}

}

with open(os.path.join(save\_dir, "software\_manifest.json"), "w", encoding="utf-8") as f:

json.dump(manifest, f, ensure\_ascii=False, indent=2)

with open(os.path.join(save\_dir, "hyperparameter\_grids.json"), "w", encoding="utf-8") as f:

json.dump(param\_grids, f, ensure\_ascii=False, indent=2)

def \_deterministic\_run\_tag(prefix: str = "run") -> str:

"""

生成确定性 run tag：基于 (GLOBAL\_SEED + 当前时间到秒) 的哈希，避免随机数。

"""

ts = datetime.datetime.now().strftime("%Y%m%d%H%M%S")

s = f"{GLOBAL\_SEED}-{ts}"

h = hashlib.md5(s.encode("utf-8")).hexdigest()[:8]

return f"{prefix}\_{ts}\_{h}"

def ML\_Classification\_Survival(

df\_seer,

df\_chinese,

features,

survival\_col,

event\_col,

prediction\_horizons=[0.5, 1, 3, 5],

decimal\_num=3,

scoring='roc\_auc',

n\_splits=10,

explain=True,

explain\_numvar=5,

explain\_sample=2,

searching=True,

savePath=None,

dpi=600,

picFormat='jpeg',

modelSave=True,

randomState=GLOBAL\_SEED,

manifest\_dir: str = "./supplementary\_outputs/",

\*\*kwargs,

):

"""

基于生存数据的多时间点机器学习分类分析

"""

# 导出 manifest 与 超参网格

param\_grids = get\_hyperparameter\_grids()

\_export\_manifest\_and\_grids(manifest\_dir, param\_grids, n\_splits)

colors = x5.CB91\_Grad\_BP

str\_time = \_deterministic\_run\_tag("mlsurv")

# 数据划分

data\_splits = split\_datasets(df\_seer, df\_chinese, features, survival\_col, event\_col,

test\_ratio=0.2, random\_state=randomState)

# 获取5种机器学习算法和参数网格

algorithms = get\_ml\_algorithms()

# 存储结果

results\_dict = {'str\_result': {}, 'tables': {}, 'pics': {}, 'save\_pics': {}, 'models': {}}

str\_result = f"采用5种机器学习方法（Random Forest, Logistic Regression, XGBoost, Decision Tree, GBDT）进行多时间点生存预测分析\n"

str\_result += f"预测时间点包括：{', '.join([str(h) for h in prediction\_horizons])}年\n"

str\_result += f"模型特征包括：{', '.join(features)}\n"

str\_result += f"数据集划分：SEER训练集N={data\_splits['train']['features'].shape[0]}例，"

str\_result += f"SEER内部测试集N={data\_splits['internal\_test']['features'].shape[0]}例，"

str\_result += f"中国医院外部验证集N={data\_splits['external\_test']['features'].shape[0]}例\n"

str\_result += f"参数优化方法：{n\_splits}折交叉验证 + 网格搜索（每折训练集内进行 Cox 单、多因素特征筛选，p<0.05 入模）\n\n"

# 对每个时间点和每种算法进行建模

for horizon in prediction\_horizons:

str\_result += f"=== {horizon}年预测结果 ===\n"

# 为当前时间点创建标签和权重

train\_labels\_info = create\_binary\_labels\_multiple\_horizons(

data\_splits['train']['survival\_times'],

data\_splits['train']['event\_indicators'],

[horizon]

)[f'{horizon}year']

internal\_labels\_info = create\_binary\_labels\_multiple\_horizons(

data\_splits['internal\_test']['survival\_times'],

data\_splits['internal\_test']['event\_indicators'],

[horizon]

)[f'{horizon}year']

external\_labels\_info = create\_binary\_labels\_multiple\_horizons(

data\_splits['external\_test']['survival\_times'],

data\_splits['external\_test']['event\_indicators'],

[horizon]

)[f'{horizon}year']

# 获取有效样本

X\_train = data\_splits['train']['features'].iloc[train\_labels\_info['indices']]

y\_train = train\_labels\_info['labels']

w\_train = train\_labels\_info['weights']

X\_internal\_test = data\_splits['internal\_test']['features'].iloc[internal\_labels\_info['indices']]

y\_internal\_test = internal\_labels\_info['labels']

w\_internal\_test = internal\_labels\_info['weights']

X\_external\_test = data\_splits['external\_test']['features'].iloc[external\_labels\_info['indices']]

y\_external\_test = external\_labels\_info['labels']

w\_external\_test = external\_labels\_info['weights']

# 对每种算法进行建模

horizon\_results = {}

best\_params\_summary = {}

for alg\_name, base\_alg in algorithms.items():

print(f"训练 {alg\_name} for {horizon}年预测...")

# 网格搜索 + StratifiedKFold（每折训练集内做 Cox 特征筛选）

if searching:

best\_model, best\_params, cv\_score = perform\_grid\_search\_with\_cv(

algorithm=base\_alg,

param\_grid=param\_grids[alg\_name],

X\_train=X\_train,

y\_train=y\_train,

sample\_weights=w\_train,

cv=n\_splits,

scoring=scoring,

cox\_times\_full=data\_splits['train']['survival\_times'],

cox\_events\_full=data\_splits['train']['event\_indicators'],

cox\_p\_univ=0.10,

cox\_p\_multiv=0.05,

cox\_max\_vars=None,

cox\_penalizer=0.0,

cox\_robust=True

)

clf = best\_model

best\_params\_summary[alg\_name] = best\_params

else:

cox\_selector = CoxSelector(

survival\_times\_full=data\_splits['train']['survival\_times'],

event\_indicators\_full=data\_splits['train']['event\_indicators'],

p\_univ=0.10, p\_multiv=0.05, max\_vars=None, penalizer=0.0, robust=True

)

pipe = Pipeline([

('cox\_sel', cox\_selector),

('clf', base\_alg)

])

pipe.fit(X\_train, y\_train, \*\*{'clf\_\_sample\_weight': w\_train})

clf = pipe

cv\_score = None

# 预测

y\_pred\_internal = clf.predict\_proba(X\_internal\_test)[:, 1]

y\_pred\_external = clf.predict\_proba(X\_external\_test)[:, 1]

# 计算加权指标

internal\_auc, internal\_brier = calculate\_weighted\_auc\_brier(y\_internal\_test, y\_pred\_internal, w\_internal\_test)

external\_auc, external\_brier = calculate\_weighted\_auc\_brier(y\_external\_test, y\_pred\_external, w\_external\_test)

# 存储结果

horizon\_results[alg\_name] = {

'internal\_auc': internal\_auc,

'internal\_brier': internal\_brier,

'external\_auc': external\_auc,

'external\_brier': external\_brier,

'model': clf,

'cv\_score': cv\_score if searching else None

}

cv\_info = f", CV分数={cv\_score:.3f}" if searching else ""

str\_result += f"{alg\_name}: 内部测试AUC={internal\_auc:.3f}, 外部验证AUC={external\_auc:.3f}{cv\_info}\n"

# 保存模型

if modelSave:

model\_filename = f"{alg\_name}\_{horizon}year\_{str\_time}.pkl"

model\_path = os.path.join(savePath, model\_filename) if savePath else model\_filename

with open(model\_path, 'wb') as f:

pickle.dump(clf, f)

results\_dict['models'][f"{alg\_name}\_{horizon}year"] = model\_filename

# 记录最优参数

if searching:

str\_result += f"\n{horizon}年预测最优参数:\n"

for alg\_name, params in best\_params\_summary.items():

str\_result += f"{alg\_name}: {params}\n"

# 找到最佳模型

best\_model\_name = max(horizon\_results.keys(),

key=lambda x: horizon\_results[x]['external\_auc'])

best\_model = horizon\_results[best\_model\_name]['model']

str\_result += f"最佳模型：{best\_model\_name} (外部验证AUC={horizon\_results[best\_model\_name]['external\_auc']:.3f})\n\n"

# 绘制ROC曲线

if savePath:

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5), dpi=dpi)

fpr\_int, tpr\_int, \_ = roc\_curve(y\_internal\_test,

best\_model.predict\_proba(X\_internal\_test)[:, 1],

sample\_weight=w\_internal\_test)

ax1.plot(fpr\_int, tpr\_int, 'b-', lw=2,

label=f'Internal Test AUC={horizon\_results[best\_model\_name]["internal\_auc"]:.3f}')

ax1.plot([0, 1], [0, 1], 'r--', alpha=0.8)

ax1.set\_xlabel('1-Specificity')

ax1.set\_ylabel('Sensitivity')

ax1.set\_title(f'{horizon}年预测 - 内部测试集ROC ({best\_model\_name})')

ax1.legend()

ax1.grid(alpha=0.3)

fpr\_ext, tpr\_ext, \_ = roc\_curve(y\_external\_test,

best\_model.predict\_proba(X\_external\_test)[:, 1],

sample\_weight=w\_external\_test)

ax2.plot(fpr\_ext, tpr\_ext, 'g-', lw=2,

label=f'External Test AUC={horizon\_results[best\_model\_name]["external\_auc"]:.3f}')

ax2.plot([0, 1], [0, 1], 'r--', alpha=0.8)

ax2.set\_xlabel('1-Specificity')

ax2.set\_ylabel('Sensitivity')

ax2.set\_title(f'{horizon}年预测 - 外部验证集ROC ({best\_model\_name})')

ax2.legend()

ax2.grid(alpha=0.3)

plt.tight\_layout()

roc\_filename = save\_fig(savePath, f'ROC\_{horizon}year', picFormat, fig, str\_time=str\_time)

results\_dict['pics'][f'ROC\_{horizon}年'] = roc\_filename

plt.close()

# 创建结果表格

results\_df = pd.DataFrame({

'算法': list(horizon\_results.keys()),

'内部测试AUC': [f"{v['internal\_auc']:.3f}" for v in horizon\_results.values()],

'内部测试Brier': [f"{v['internal\_brier']:.3f}" for v in horizon\_results.values()],

'外部验证AUC': [f"{v['external\_auc']:.3f}" for v in horizon\_results.values()],

'外部验证Brier': [f"{v['external\_brier']:.3f}" for v in horizon\_results.values()],

'CV分数': [f"{v['cv\_score']:.3f}" if v['cv\_score'] is not None else "N/A" for v in horizon\_results.values()]

})

results\_dict['tables'][f'{horizon}年预测结果'] = results\_df

# SHAP解释（仅对最佳模型）——占位，按需补充

if explain:

print(f"为{horizon}年最佳模型({best\_model\_name})生成SHAP解释...")

try:

pass

except Exception:

pass

results\_dict['str\_result'] = str\_result

return results\_dict