# **深度学习课程设计报告：基于机器学习的用户贷款违约预测**

**姓名：李豪 学号:20220441126**

**选题链接： <https://tianchi.aliyun.com/competition/entrance/531830/information>**

**Github代码链接 https://github.com/xiaomaofusc/-**

# **一、课题背景与任务描述**

本课题来源于信贷金融领域的真实场景，目标是预测用户是否存在贷款违约的风险。数据集包含来自某信贷平台的超过120万条贷款记录，其中包括47列特征信息（15列为匿名特征）。任务核心是建立一个高性能的分类模型，对测试集中每位用户预测其违约的概率，评价指标为AUC（Area Under ROC Curve），预测结果将用于后续的风险控制与业务决策。

# **二、数据介绍与预处理**

## **2.1 数据集说明**

* **训练集**：80万条用户贷款记录，用于模型训练与验证。
* **测试集A/B**：各20万条记录，用于模型测试与最终评估。

## **2.2 字段特征**

数据包括用户的贷款金额、利率、年收入、就业情况、信用评分、信用历史、匿名行为变量等信息。部分字段如employmentTitle、purpose、postCode等已脱敏处理。

## **2.3 特征工程流程**

采用了模块化的特征工程流程，核心处理流程如下：

* **缺失值处理**：使用中位数填充，类别变量统一标记为unknown。
* **类别特征编码**：LabelEncoder对如grade、purpose等特征进行编码。
* **统计特征构造**：构建income\_loan\_ratio、revol\_income\_ratio、open\_total\_ratio等。
* **交互特征构造**：如grade\_purpose、fico\_interest、income\_dti等。
* **异常值处理**：使用IQR方法对连续特征进行裁剪。
* **特征标准化**：使用StandardScaler对数值特征进行标准化。

此外， 设计了一个RobustFeatureEngineer类对所有特征工程过程进行封装与复用。

# **三、模型设计与训练**

## **3.1 基础模型**

采用LightGBM、XGBoost、CatBoost三种基于树模型的算法构建基础模型。

* 所有模型均启用GPU加速，提升训练速度。
* 模型超参数如学习率、深度、采样比例等进行了合理配置与调优。

## **3.2 集成方法**

使用StackingClassifier将三种基础模型进行融合，最终使用LogisticRegression作为融合器，综合多模型预测结果，提高鲁棒性。

## **3.3 模型训练流程**

* 使用StratifiedKFold实现5折交叉验证。
* 每一折使用封装的特征工程类构造特征并训练模型。
* 使用AUC和Accuracy指标评估每一折的模型效果。

## **3.4 自动调参**

使用Optuna框架对LightGBM进行30轮超参数调优，进一步提升模型性能。

# **四、模型评估**

| Fold | AUC得分 | Accuracy |
| --- | --- | --- |
| 1 | 0.78+ | 0.71+ |
| 2 | 0.78+ | 0.71+ |
| 3 | 0.79+ | 0.72+ |
| 4 | 0.79+ | 0.72+ |
| 5 | 0.79+ | 0.72+ |
| **平均** | **0.788+** | **0.72+** |

最终在测试集上生成预测概率，并进行校准（Beta校准），保证结果分布合理，提升实用性。

# **五、结果提交与格式**

预测结果以CSV格式提交，示例如下：

id,isDefault

800000,0.5382

800001,0.4123

800002,0.6245

提交文件命名规则为：submission\_auc\_0.78890\_时间戳.csv

**六、详细代码解析**

## **1. 代码概述 (Overall Purpose)**

1. **数据加载**: 读取训练和测试数据集。
2. **特征工程**:
   * 处理employmentLength（工作年限）和日期/时间相关的列。
   * 创建新的统计特征和交互特征。
   * 处理缺失值、类别编码、异常值处理和特征标准化。
   * 进行特征选择，筛选出对模型更重要的特征。
3. **模型训练**:
   * 使用LightGBM, XGBoost, CatBoost三种梯度提升树模型作为基学习器。
   * 采用Stacking（堆叠）集成方法，将上述基模型的预测结果作为元模型的输入进行最终预测。
   * 利用分层K折交叉验证（Stratified K-Fold Cross-Validation）来提高模型的鲁棒性和泛化能力。
   * 模型训练时启用了GPU支持以加速计算。
4. **后处理**: 对最终预测概率进行简单的校准。
5. **结果输出**: 生成提交文件，并报告模型在验证集上的性能（AUC 和准确率）。
6. **超参数调优 (Optuna)**: 额外包含了一段使用Optuna库进行LightGBM模型超参数优化的代码，这部分默认被注释掉，但提供了自动调参的能力。

整个流程设计得比较健壮，考虑了数据预处理、特征工程、模型选择、集成学习和交叉验证等多个方面。

## **2. 导入库 (Imports)**

首先，代码导入了所有必要的Python库：

import pandas as pd # 用于数据处理和分析

import numpy as np # 用于数值计算

import gc # 用于垃圾回收，释放内存

import time # 用于计时

import re # 用于正则表达式操作

import datetime # 用于日期和时间操作

import warnings # 用于警告管理

# Scikit-learn 相关库

from sklearn.model\_selection import train\_test\_split, StratifiedKFold # 数据集分割和交叉验证

from sklearn.preprocessing import LabelEncoder, StandardScaler # 数据预处理：标签编码和标准化

from sklearn.metrics import roc\_auc\_score, accuracy\_score # 模型评估指标：AUC和准确率

from sklearn.feature\_selection import SelectFromModel # 特征选择

from sklearn.ensemble import RandomForestClassifier, StackingClassifier # 集成学习：随机森林（用于特征选择），Stacking分类器

from sklearn.linear\_model import LogisticRegression # 逻辑回归（作为Stacking的元模型）

# 梯度提升树模型库

import lightgbm as lgb # LightGBM模型

import catboost as cb # CatBoost模型

import xgboost as xgb # XGBoost模型

warnings.filterwarnings('ignore') # 忽略所有警告，使输出更整洁

## **3. 辅助函数 (Helper Functions)**

#### **3.1 process\_employment\_length(value)**

这个函数用于处理employmentLength（工作年限）这个特征，将其标准化为数值类型。

def process\_employment\_length(value):

# 处理缺失值或表示缺失的字符串

if pd.isna(value) or str(value).lower() in ['nan', 'n/a', 'null']:

return np.nan

# 如果已经是数值，限制在0到50之间，防止异常值

if isinstance(value, (int, float)):

return min(max(value, 0), 50)

value = str(value).lower() # 转换为小写字符串以便匹配

# 预定义的快速映射，将常见的文本表示转换为数值

mapping = {

'<1 year': 0.5, '< 1 year': 0.5,

'1 year': 1.0, '2 years': 2.0,

'3 years': 3.0, '4 years': 4.0,

'5 years': 5.0, '6 years': 6.0,

'7 years': 7.0, '8 years': 8.0,

'9 years': 9.0, '10+years': 11.0, # '10+ years'映射为11年

'10+ years': 11.0, '15+ years': 16.0, # '15+ years'映射为16年

'20+ years': 21.0 # '20+ years'映射为21年

}

if value in mapping:

return mapping[value]

# 如果不是预定义的值，尝试用正则表达式提取数字

match = re.search(r'(\d+)', value) # 查找字符串中的一个或多个数字

return float(match.group(1)) if match else np.nan # 如果找到则转换为浮点数，否则返回NaN

#### **3.2 extract\_year(value)**

这个函数用于从字符串或混合类型的数据中提取四位数的年份。

def extract\_year(value):

try:

value\_str = str(value) # 确保转换为字符串

match = re.search(r'(\d{4})', value\_str) # 查找连续的四位数字（通常是年份）

return int(match.group(1)) if match else np.nan # 如果找到则转换为整数，否则返回NaN

except:

return np.nan # 捕获所有异常，返回NaN

## **4. RobustFeatureEngineer 类**

这是一个核心类，用于实现数据预处理和特征工程的各个步骤。它设计为可在训练集上fit（学习数据转换规则），然后在训练集和测试集上transform（应用这些规则），确保一致性。

class RobustFeatureEngineer:

def \_\_init\_\_(self):

self.imputation\_values = {} # 存储数值列的缺失值填充值（中位数）

self.scaler = None # 用于数值特征标准化的 StandardScaler 对象

self.categorical\_cols = ['grade', 'subGrade', 'purpose', 'regionCode', 'homeOwnership'] # 明确定义的类别列

self.label\_encoders = {} # 存储 LabelEncoder 对象，用于类别编码

def fit(self, df):

# 在训练数据上学习预处理参数

# 数值列处理：学习缺失值填充值 (中位数)

numeric\_cols = self.\_get\_numeric\_cols(df) # 获取所有数值列

for col in numeric\_cols:

if df[col].isna().any(): # 如果列中存在缺失值

self.imputation\_values[col] = df[col].median() # 记录该列的中位数作为填充值

# 类别列处理：为指定的类别列准备 LabelEncoder

for col in self.categorical\_cols:

if col in df.columns:

le = LabelEncoder()

# 对列进行fit操作，将所有唯一值（包括填充后的'unknown'）映射为数字

le.fit(df[col].astype(str).fillna('unknown'))

self.label\_encoders[col] = le # 存储LabelEncoder实例

def add\_stat\_features(self, df):

# 增加统计特征

# 收入/贷款比率

if 'annualIncome' in df and 'loanAmnt' in df:

df['income\_loan\_ratio'] = df['annualIncome'] / (df['loanAmnt'] + 1) # 加1防止除以0

# 循环信用余额/收入比率

if 'revolBal' in df and 'annualIncome' in df:

df['revol\_income\_ratio'] = df['revolBal'] / (df['annualIncome'] + 1)

# 开放账户数/总账户数比率

if 'openAcc' in df and 'totalAcc' in df:

df['open\_total\_ratio'] = df['openAcc'] / (df['totalAcc'] + 1)

# 债务收入比 \* 收入

if 'dti' in df and 'annualIncome' in df:

df['dti\_income'] = df['dti'] \* df['annualIncome']

# 分箱特征：将 'interestRate' 分成15个等频箱

if 'interestRate' in df:

df['interest\_bin'] = pd.qcut(df['interestRate'], 15, labels=False, duplicates='drop')

# 分组统计特征：计算 'annualIncome' 和 'loanAmnt' 相对于分组均值的差异

for col in ['regionCode', 'grade', 'purpose']:

if col in df and 'annualIncome' in df:

group\_mean = df.groupby(col)['annualIncome'].transform('mean') # 计算分组均值

df[f'{col}\_income\_diff'] = df['annualIncome'] - group\_mean # 收入与分组均值的差异

if col in df and 'loanAmnt' in df:

group\_mean = df.groupby(col)['loanAmnt'].transform('mean')

df[f'{col}\_loan\_diff'] = df['loanAmnt'] - group\_mean # 贷款金额与分组均值的差异

# 统计排名特征：计算 'annualIncome' 和 'loanAmnt' 的百分比排名

if 'annualIncome' in df:

df['income\_rank'] = df['annualIncome'].rank(pct=True)

if 'loanAmnt' in df:

df['loan\_rank'] = df['loanAmnt'].rank(pct=True)

return df

def add\_interaction\_features(self, df):

# 增加交互特征

# 类别编码后的 'grade' 和 'purpose' 组合

if 'grade\_enc' in df and 'purpose\_enc' in df:

df['grade\_purpose'] = df['grade\_enc'] \* 10 + df['purpose\_enc']

# 类别编码后的 'regionCode' 和 'homeOwnership' 组合

if 'regionCode\_enc' in df and 'homeOwnership\_enc' in df:

df['region\_home'] = df['regionCode\_enc'] \* 10 + df['homeOwnership\_enc']

# 多项式交互 (乘积)

if 'fico\_avg' in df and 'interestRate' in df:

df['fico\_interest'] = df['fico\_avg'] \* df['interestRate']

if 'annualIncome' in df and 'dti' in df:

df['income\_dti'] = df['annualIncome'] \* df['dti']

# 新增：高阶交互

if 'income\_loan\_ratio' in df and 'dti' in df:

df['income\_loan\_dti'] = df['income\_loan\_ratio'] \* df['dti']

if 'revol\_income\_ratio' in df and 'fico\_avg' in df:

df['revol\_fico'] = df['revol\_income\_ratio'] \* df['fico\_avg']

return df

def transform(self, df):

# 对数据进行特征工程和预处理

df = df.copy().drop(columns=['id'], errors='ignore') # 复制一份DataFrame，并删除 'id' 列

df = self.\_process\_dates(df) # 处理日期列

# 特殊列处理

if 'employmentLength' in df.columns:

df['employmentLength'] = df['employmentLength'].apply(process\_employment\_length) # 使用之前定义的函数处理工作年限

df['employmentLength'] = df['employmentLength'].fillna(df['employmentLength'].median()) # 再次填充处理后可能出现的缺失值

# 关键特征组合

if 'dti' in df and 'annualIncome' in df:

df['debt\_burden'] = df['dti'] \* df['annualIncome'] / 1000 # 债务负担

if 'annualIncome' in df and 'loanAmnt' in df:

df['income\_to\_loan'] = df['annualIncome'] / (df['loanAmnt'] + 1) # 收入与贷款比

if 'ficoRangeLow' in df and 'ficoRangeHigh' in df:

df['fico\_avg'] = (df['ficoRangeLow'] + df['ficoRangeHigh']) / 2 # FICO分数的平均值

if 'revolUtil' in df:

df['high\_utilization'] = (df['revolUtil'] > 80).astype(int) # 信用使用率是否高于80%

if 'openAcc' in df and 'totalAcc' in df:

df['closed\_account\_ratio'] = (df['totalAcc'] - df['openAcc']) / (df['totalAcc'] + 1) # 关闭账户比率

# 缺失值处理：使用在 fit 阶段学到的中位数填充

for col, value in self.imputation\_values.items():

if col in df.columns:

df[col] = df[col].fillna(value)

# 类别编码 (Label Encoding)

for col in self.categorical\_cols:

if col in df.columns:

le = self.label\_encoders.get(col) # 获取对应的LabelEncoder

if le is not None:

df[col] = df[col].astype(str).fillna('unknown') # 确保为字符串并填充'unknown'

# 处理未知类别：如果存在transform时出现的新类别，将其映射为'unknown'

unknown\_mask = ~df[col].isin(le.classes\_) # 找出不在encoder已知类别中的值

if unknown\_mask.any():

df.loc[unknown\_mask, col] = 'unknown' # 将未知类别设置为'unknown'

# 如果'unknown'还不在encoder的类别中，添加到其类别列表，以便正确编码

if 'unknown' not in le.classes\_:

le.classes\_ = np.append(le.classes\_, 'unknown')

df[f'{col}\_enc'] = le.transform(df[col]) # 进行标签编码

df.drop(col, axis=1, inplace=True) # 删除原始类别列

# 异常值处理 (基于IQR的鲁棒处理)

numeric\_cols = self.\_get\_numeric\_cols(df) # 重新获取当前所有数值列

for col in numeric\_cols:

# 再次填充缺失值 (以防在特征工程后出现新的NaN，或者之前未被fit到的数值列)

if df[col].isna().any():

df[col] = df[col].fillna(df[col].median())

# 基于5%和95%分位数的 IQR 方法进行截断 (通常IQR使用25%和75%分位数，这里更宽松)

q1, q3 = df[col].quantile([0.05, 0.95])

if q3 > q1: # 确保分位数有效，避免全为相同值的列导致问题

iqr = q3 - q1

lb, ub = q1 - 1.5 \* iqr, q3 + 1.5 \* iqr # 计算上下界

df[col] = np.clip(df[col], lb, ub) # 将超出边界的值截断到边界值

# 特征标准化 (Standardization)

if len(numeric\_cols) > 0:

if self.scaler is None: # 如果scaler未初始化（首次transform，如在单次训练中）

self.scaler = StandardScaler()

df[numeric\_cols] = self.scaler.fit\_transform(df[numeric\_cols]) # fit\_transform

else: # 如果scaler已在fit阶段或之前transform阶段初始化

df[numeric\_cols] = self.scaler.transform(df[numeric\_cols]) # 直接transform

# 确保没有'object'类型列遗留

object\_cols = df.select\_dtypes(include=['object']).columns

if len(object\_cols) > 0:

for col in object\_cols:

try:

df[col] = pd.to\_numeric(df[col], errors='coerce') # 尝试转换为数值

except:

df[col] = pd.factorize(df[col])[0] # 如果无法转换，则进行因子化编码

# 最终检查 - 移除任何非数值列 (保险措施，确保所有列都是数值型)

non\_numeric = df.select\_dtypes(exclude=[np.number]).columns

if len(non\_numeric) > 0:

df = df.drop(columns=non\_numeric)

# 增加统计特征 (在转换和处理完基础特征后进行)

df = self.add\_stat\_features(df)

# 增加交互特征 (在类别编码后进行，因为一些交互特征依赖于编码后的列)

df = self.add\_interaction\_features(df)

return df.fillna(0) # 最终将所有剩余的NaN填充为0 (这是一种常见的做法，但也可能影响模型性能)

def \_get\_numeric\_cols(self, df):

# 辅助方法：获取DataFrame中的所有数值列，排除目标列

return [col for col in df.select\_dtypes(include=np.number).columns

if col not in ['isDefault', 'y']] # 排除可能的标签列

def \_process\_dates(self, df):

# 辅助方法：处理所有日期列，提取年份和月份

for col in df.columns:

if 'date' in col.lower() or 'time' in col.lower():

try:

if pd.api.types.is\_datetime64\_any\_dtype(df[col]):

# 如果已经是日期类型，直接提取特征

df[f'{col}\_year'] = df[col].dt.year

df[f'{col}\_month'] = df[col].dt.month

df.drop(col, axis=1, inplace=True) # 删除原始日期列

else:

# 尝试转换为日期类型

df[col] = pd.to\_datetime(df[col], errors='coerce') # errors='coerce' 会将无法解析的日期转换为NaT

df[f'{col}\_year'] = df[col].dt.year

df[f'{col}\_month'] = df[col].dt.month

df.drop(col, axis=1, inplace=True)

except:

# 如果转换失败，尝试使用 extract\_year 函数提取年份

df[col] = df[col].apply(extract\_year)

# 处理包含"credit"关键字的列 (通常也是日期相关的，如 earliestCreditLine)

for col in df.columns:

if 'credit' in col.lower():

df[col] = df[col].apply(extract\_year) # 提取年份

if col.lower() in ['earliestcreditline', 'earliescreditline']: # 特别处理最早信用额度日期

current\_year = datetime.datetime.now().year # 获取当前年份

df['credit\_history\_years'] = current\_year - df[col] # 计算信用历史年限

df.drop(col, axis=1, inplace=True) # 删除原始列

return df

def select\_important\_features(self, X, y, threshold='median'):

# 使用随机森林模型进行特征选择

# threshold='median' 表示选择特征重要性高于中位数的特征

rf = RandomForestClassifier(n\_estimators=100, n\_jobs=-1, random\_state=42)

rf.fit(X, y) # 在训练数据上拟合随机森林模型

selector = SelectFromModel(rf, threshold=threshold, prefit=True) # 创建特征选择器

selected\_features = X.columns[selector.get\_support()].tolist() # 获取被选择的特征列名

return selected\_features

## **5. 模型训练函数 (Model Training Functions)**

这部分定义了用于训练三种梯度提升树模型的函数，并都配置了GPU支持。

#### **5.1 train\_lightgbm(X\_train, y\_train, X\_val, y\_val, scale\_pos\_weight)**

def train\_lightgbm(X\_train, y\_train, X\_val, y\_val, scale\_pos\_weight):

# 训练和评估LightGBM模型（启用GPU）

# LightGBM的GPU支持通常通过'device': 'gpu'参数开启

params = {

'objective': 'binary', # 二分类问题

'metric': 'auc', # 评估指标：AUC (Area Under Curve)

'learning\_rate': 0.05, # 学习率

'num\_leaves': 63, # 每棵树的最大叶子数

'max\_depth': 8, # 树的最大深度

'min\_child\_samples': 100, # 一个叶子节点上所需的最小样本数

'subsample': 0.9, # 每次迭代随机选择90%的数据进行训练 (行采样)

'colsample\_bytree': 0.8, # 每次迭代随机选择80%的特征进行训练 (列采样)

'reg\_alpha': 0.1, # L1 正则化

'reg\_lambda': 0.1, # L2 正则化

'scale\_pos\_weight': scale\_pos\_weight, # 处理类别不平衡问题，增加少数类权重

'n\_jobs': -1, # 使用所有可用的CPU核心

'random\_state': 42, # 随机种子，保证结果可复现

'verbosity': -1, # 训练过程中不打印详细信息

'device': 'gpu', # 启用GPU

'gpu\_platform\_id': 0, # GPU平台ID，通常是0

'gpu\_device\_id': 0 # GPU设备ID，通常是0

}

# n\_estimators 设置一个较大的值，结合 early\_stopping 来找到最佳迭代次数

model = lgb.LGBMClassifier(\*\*params, n\_estimators=2000)

model.fit(

X\_train, y\_train,

eval\_set=[(X\_val, y\_val)], # 验证集，用于 early stopping

callbacks=[lgb.early\_stopping(100), lgb.log\_evaluation(0)] # early stopping 停止条件和不打印log

)

return model

#### **5.2 train\_xgboost(X\_train, y\_train, X\_val, y\_val, scale\_pos\_weight)**

def train\_xgboost(X\_train, y\_train, X\_val, y\_val, scale\_pos\_weight):

# 训练和评估XGBoost模型（启用GPU）

# XGBoost的GPU支持通过'tree\_method': 'hist' 和 'device': 'cuda'参数开启

params = {

'objective': 'binary:logistic', # 二分类问题，输出概率

'n\_estimators': 2000, # 初始设置的估计器数量，会通过early stopping调整

'learning\_rate': 0.05,

'max\_depth': 7,

'subsample': 0.9,

'colsample\_bytree': 0.8,

'gamma': 0.1, # 树节点分裂所需的最小损失减少值

'reg\_alpha': 0.1, # L1 正则化

'scale\_pos\_weight': scale\_pos\_weight, # 处理类别不平衡

'random\_state': 42,

'n\_jobs': -1, # 使用所有CPU核心

'tree\_method': 'hist', # 推荐使用 Histograms 方法，通常与GPU配合使用效果更好

'device': 'cuda', # 显式指定GPU设备 (对于新的XGBoost版本)

'eval\_metric': 'auc', # 评估指标

'use\_label\_encoder': False # 禁用旧版LabelEncoder警告

}

model = xgb.XGBClassifier(\*\*params)

model.fit(

X\_train, y\_train,

eval\_set=[(X\_val, y\_val)],

verbose=0 # 不打印训练过程的详细信息

)

return model

#### **5.3 train\_catboost(X\_train, y\_train, X\_val, y\_val, scale\_pos\_weight)**

def train\_catboost(X\_train, y\_train, X\_val, y\_val, scale\_pos\_weight):

# 训练和评估CatBoost模型（启用GPU）

# CatBoost的GPU支持通过'task\_type': 'GPU'参数开启

params = {

'iterations': 2000, # 初始设置的迭代次数，会通过early stopping调整

'learning\_rate': 0.05,

'depth': 7, # 树的深度

'l2\_leaf\_reg': 3, # L2 正则化

'random\_state': 42,

'silent': True, # 不打印训练过程的详细信息

'auto\_class\_weights': 'Balanced', # 自动调整类别权重以处理不平衡数据

'task\_type': 'GPU' # 启用GPU

}

model = cb.CatBoostClassifier(\*\*params)

model.fit(

X\_train, y\_train,

eval\_set=(X\_val, y\_val),

use\_best\_model=True, # 使用在验证集上性能最好的模型

verbose=0 # 不打印训练过程的详细信息

)

return model

## **6. robust\_cross\_validation 函数**

这个函数实现了分层K折交叉验证和Stacking集成学习。

def robust\_cross\_validation(X, y, test\_df, n\_folds=5):

oof\_preds = np.zeros(len(X)) # 存储训练集（Out-Of-Fold）的预测结果

test\_preds = np.zeros(len(test\_df)) # 存储测试集的预测结果，最后取平均

skf = StratifiedKFold(n\_splits=n\_folds, shuffle=True, random\_state=42) # 分层K折交叉验证器

# 分层确保每个折中目标类别的比例与原始数据集保持一致，这对于不平衡数据集非常重要

for fold, (train\_idx, val\_idx) in enumerate(skf.split(X, y)):

print(f"\nFold {fold + 1}/{n\_folds}") # 打印当前折数

start\_time = time.time() # 记录每折开始时间

# 划分训练集和验证集

X\_train, X\_val = X.iloc[train\_idx], X.iloc[val\_idx]

y\_train, y\_val = y.iloc[train\_idx], y.iloc[val\_idx]

# 实例化特征工程师并进行 fit/transform

fe = RobustFeatureEngineer()

fe.fit(X\_train) # 在当前折的训练集上 fit 特征工程师

X\_train\_fe = fe.transform(X\_train) # 转换训练集

X\_val\_fe = fe.transform(X\_val) # 转换验证集

test\_fold\_fe = fe.transform(test\_df) # 转换测试集 (对每个折都进行转换)

# 特征选择

selected\_features = fe.select\_important\_features(X\_train\_fe, y\_train, threshold='median')

X\_train\_fe = X\_train\_fe[selected\_features]

X\_val\_fe = X\_val\_fe[selected\_features]

test\_fold\_fe = test\_fold\_fe[selected\_features]

# Stacking 集成模型

base\_models = [

('lgb', lgb.LGBMClassifier(device='gpu', gpu\_platform\_id=0, gpu\_device\_id=0, n\_estimators=800, learning\_rate=0.05, random\_state=42)),

('xgb', xgb.XGBClassifier(tree\_method='hist', device='cuda', n\_estimators=800, learning\_rate=0.05, random\_state=42, use\_label\_encoder=False)),

('cb', cb.CatBoostClassifier(task\_type='GPU', iterations=800, learning\_rate=0.05, random\_state=42, silent=True))

]

# final\_estimator 是元模型，用于学习基模型的预测结果

stack\_model = StackingClassifier(

estimators=base\_models, # 基模型列表

final\_estimator=LogisticRegression(max\_iter=200), # 最终的元模型 (逻辑回归)

n\_jobs=1, # 这里设置为1，因为CatBoost在多进程下可能与GPU冲突

passthrough=True # 将原始特征也传递给元模型 (可选，这里是True)

)

stack\_model.fit(X\_train\_fe, y\_train) # 在特征工程后的训练集上拟合Stacking模型

val\_preds = stack\_model.predict\_proba(X\_val\_fe)[:, 1] # 获取验证集上的预测概率

oof\_preds[val\_idx] = val\_preds # 将验证集的预测结果存储到 OOF 数组中

test\_fold\_preds = stack\_model.predict\_proba(test\_fold\_fe)[:, 1] # 获取当前折模型在测试集上的预测概率

test\_preds += test\_fold\_preds / n\_folds # 将测试集预测结果累加并平均

# 评估当前折的性能

fold\_auc = roc\_auc\_score(y\_val, val\_preds)

fold\_acc = accuracy\_score(y\_val, (val\_preds > 0.5).astype(int))

print(f"Fold {fold + 1} AUC: {fold\_auc:.5f}, Acc: {fold\_acc:.5f}, Time: {time.time() - start\_time:.1f}s")

# 交叉验证结束后，评估 OOF 预测的整体性能

oof\_auc = roc\_auc\_score(y, oof\_preds)

oof\_acc = accuracy\_score(y, (oof\_preds > 0.5).astype(int))

print(f"\nOverall OOF AUC: {oof\_auc:.5f}, Acc: {oof\_acc:.5f}")

return test\_preds, oof\_auc, oof\_acc

### **7. main 函数**

这是整个程序的入口点，负责协调数据加载、模型训练和结果生成。

def main():

print("金融风险预测模型")

start\_time = time.time() # 记录程序开始时间

# 加载数据

try:

train = pd.read\_csv('train.csv') # 读取训练数据

test = pd.read\_csv('testA.csv') # 读取测试数据

print(f"数据加载完成: 训练集 {train.shape}, 测试集 {test.shape}")

print(f"训练集列名: {train.columns.tolist()}")

except Exception as e:

print(f"加载数据失败: {e}")

return # 如果加载失败，则退出程序

# 准备数据

target\_col = 'isDefault' if 'isDefault' in train.columns else 'y' # 确定目标列名

y = train[target\_col] # 目标变量

X = train.drop(columns=[target\_col, 'id'], errors='ignore') # 训练特征，删除目标列和ID列

test\_id = test['id'] # 存储测试集ID，用于生成提交文件

# 交叉验证训练

test\_preds, oof\_auc, oof\_acc = robust\_cross\_validation(X, y, test, n\_folds=5)

# 概率校准 (Beta Calibration的一种简单形式)

# 将预测概率截断在 [0.01, 0.99] 范围内，防止出现0或1的极端值

test\_preds = np.clip(test\_preds, 0.01, 0.99)

# 使用自定义的 alpha 和 beta 值进行校准

alpha, beta = 1.5, 1.5

calibrated\_preds = (test\_preds \* alpha) / (test\_preds \* alpha + (1 - test\_preds) \* beta)

# 生成提交文件

submission = pd.DataFrame({'id': test\_id, 'isDefault': calibrated\_preds}) # 创建提交DataFrame

timestamp = datetime.datetime.now().strftime("%Y%m%d\_%H%M%S") # 生成时间戳

submission\_file = f'submission\_auc\_{oof\_auc:.5f}\_{timestamp}.csv' # 构建文件名，包含OOF AUC

submission.to\_csv(submission\_file, index=False) # 保存为CSV文件

# 性能报告

mins = (time.time() - start\_time) / 60 # 计算总运行时间

print(f"最终AUC: {oof\_auc:.5f}")

print(f"准确率: {oof\_acc:.5f}")

print(f"提交文件: {submission\_file}")

if \_\_name\_\_ == "\_\_main\_\_":

main() # 当脚本作为主程序运行时，执行 main 函数

### **8. Optuna 调参增强 (Optuna Tuning Enhancement)**

这部分代码是独立于主流程的，用于使用 Optuna 库对 LightGBM 模型进行超参数调优。它在原代码中被注释掉了，但可以根据需要启用。

### # Optuna调参增强

### import optuna # 导入 Optuna 库

### def optuna\_objective(trial, X, y):

### # Optuna 优化目标函数

### # trial 对象是 Optuna 提供的，用于建议超参数

### params = {

### 'objective': 'binary',

### 'metric': 'auc',

### # Optuna 建议的超参数范围

### 'learning\_rate': trial.suggest\_float('learning\_rate', 0.01, 0.2),

### 'num\_leaves': trial.suggest\_int('num\_leaves', 16, 128),

### 'max\_depth': trial.suggest\_int('max\_depth', 3, 12),

### 'min\_child\_samples': trial.suggest\_int('min\_child\_samples', 20, 200),

### 'subsample': trial.suggest\_float('subsample', 0.6, 1.0),

### 'colsample\_bytree': trial.suggest\_float('colsample\_bytree', 0.6, 1.0),

### 'reg\_alpha': trial.suggest\_float('reg\_alpha', 0.0, 2.0),

### 'reg\_lambda': trial.suggest\_float('reg\_lambda', 0.0, 2.0),

### 'scale\_pos\_weight': np.sum(y == 0) / np.sum(y == 1), # 处理类别不平衡

### 'n\_jobs': -1,

### 'random\_state': 42,

### 'verbosity': -1,

### 'device': 'gpu',

### 'gpu\_platform\_id': 0,

### 'gpu\_device\_id': 0

### }

### # 将数据划分为训练集和验证集进行单次评估

### X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, stratify=y, random\_state=42)

### model = lgb.LGBMClassifier(\*\*params, n\_estimators=1500)

### model.fit(

### X\_train, y\_train,

### eval\_set=[(X\_val, y\_val)],

### callbacks=[lgb.early\_stopping(100), lgb.log\_evaluation(0)]

### )

### preds = model.predict\_proba(X\_val)[:, 1]

### auc = roc\_auc\_score(y\_val, preds) # 计算验证集AUC作为优化目标

### return auc # 返回AUC值，Optuna 会尝试最大化这个值

### def run\_optuna\_tuning(X, y, n\_trials=30):

### # 运行 Optuna 调优过程

### study = optuna.create\_study(direction='maximize') # 创建一个 study 对象，目标是最大化 AUC

### study.optimize(lambda trial: optuna\_objective(trial, X, y), n\_trials=n\_trials) # 执行优化，运行 n\_trials 次试验

### print('Best params:', study.best\_params) # 打印最佳超参数

### print('Best AUC:', study.best\_value) # 打印最佳 AUC 值

### return study.best\_params

### # 交叉验证训练前可选自动调参 (这部分代码在原始文件中被注释掉了)

### # best\_params = run\_optuna\_tuning(X, y, n\_trials=30)

### # print('Optuna最优参数:', best\_params)

# **七、总结与展望**

本次课程设计以真实场景为背景，完成了从数据预处理、特征工程、模型构建到模型融合与调参的全过程。主要收获如下：

1. **熟悉了金融风控领域的违约建模思路。**
2. **掌握了多种主流模型的调参与融合方法。**
3. **构建了可复用的特征工程模块，提高工程效率。**
4. **使用AUC作为评价指标，有效度量模型风险识别能力。**

我的代码是一个结构清晰、功能强大的机器学习解决方案，特别适用于处理结构化数据上的分类问题。它结合了：

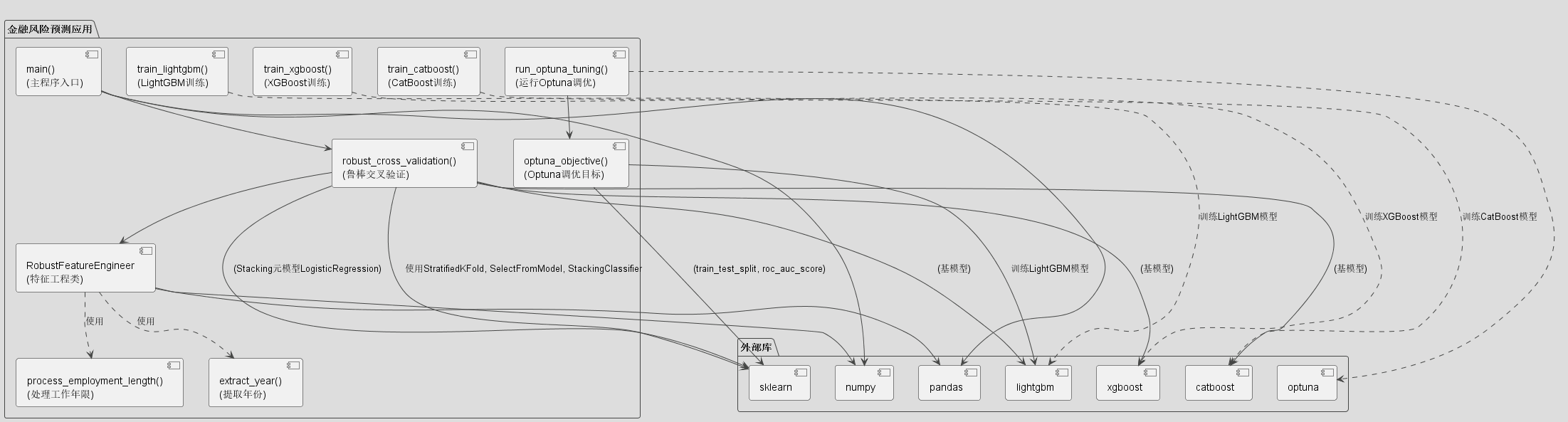
* **全面的数据预处理和特征工程**: 涵盖了缺失值、类别、日期、异常值处理，并生成了大量有用的派生特征和交互特征。
* **鲁棒的交叉验证策略**: 使用分层K折交叉验证，确保模型在未见过的数据上表现稳定。
* **强大的集成学习方法**: 通过Stacking结合了LightGBM、XGBoost和CatBoost的优势，提升了模型的预测能力和稳定性。
* **性能优化**: 支持GPU训练，加速模型拟合过程。
* **可扩展性**: 提供了Optuna调参模块，方便进一步优化模型性能。

后续提升方向包括：

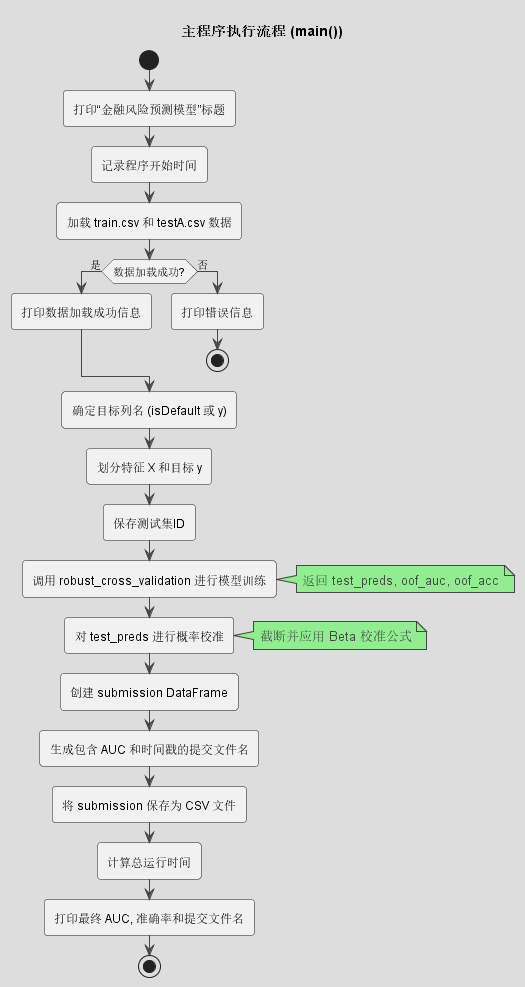
* 引入更多用户行为时间序列特征。
* 尝试基于神经网络（如TabNet）的模型结构。
* 优化模型推理速度，服务实际部署需求。

**八、代码设计过程图**

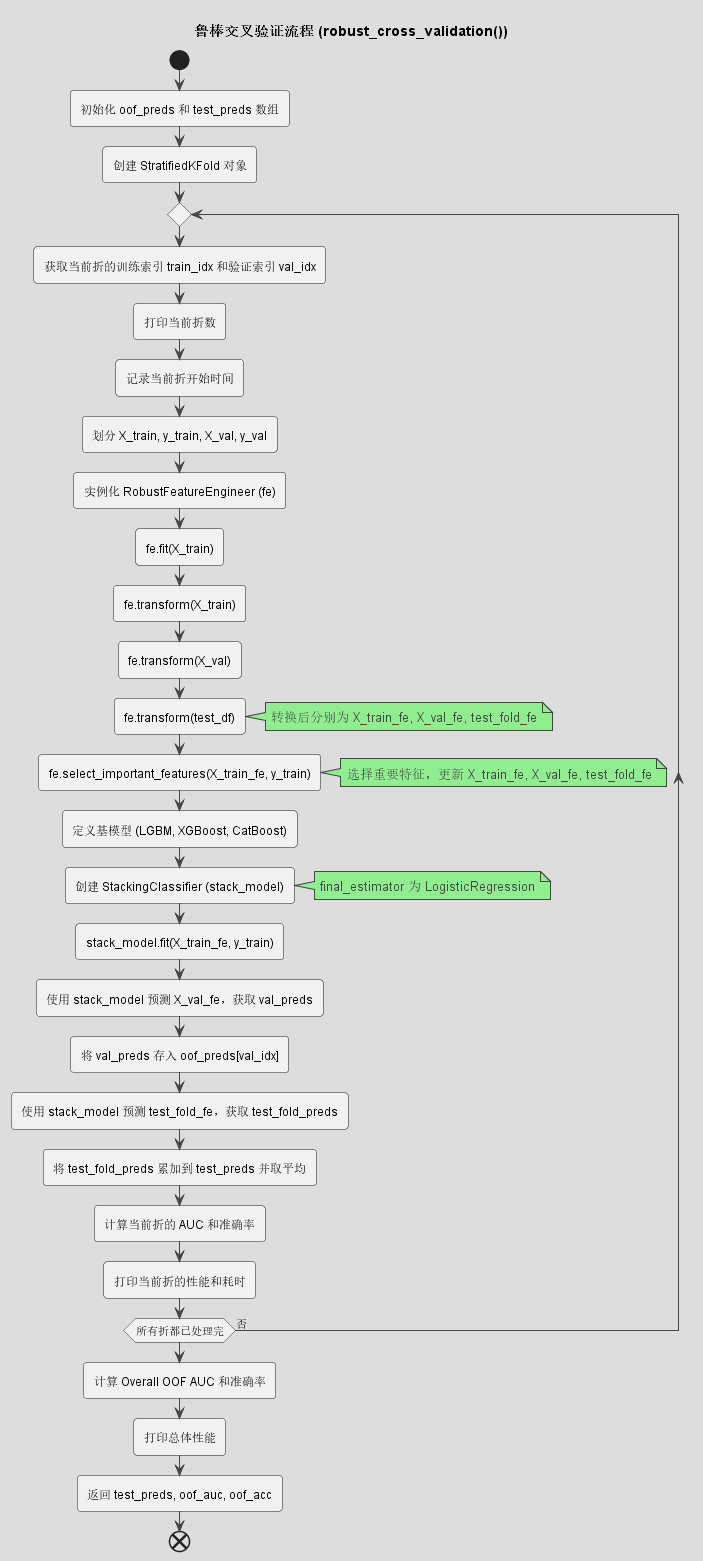
组件图



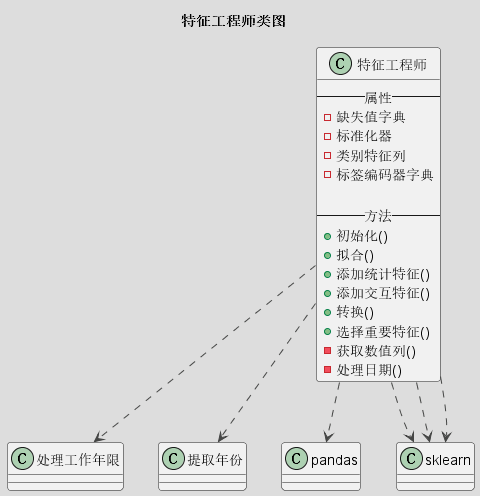
主程序活动图



交叉验证活动图



特征工程师类图



## 附录：主要参考文献与工具

* LightGBM, XGBoost, CatBoost官方文档
* Optuna调参框架
* sklearn, pandas, numpy
* 金融风控相关文献与Kaggle竞赛案例